Technical University of Crete

School of Electronic and Computer Engineering

Parallel Skyline Computation on Map - Reduce Systems

Stella Maropaki

Thesis Committee

Assistant Professor Antonios Deligiannakis (ECE)

Professor Minos Garofalakis (ECE)

Assistant Professor Vasilis Samoladas (ECE)

Chania, December 2013

Πολυτεχνειο Κρητης

Σχολη Ηλεκτρονικων Μηχανικων και Μηχανικων Υπολογιστων

Παράλληλος Υπολογισμός Κορυφογραμμής σε Σύστημα Απεικόνισης - Ελάττωσης

Στέλλα Μαροπάκη

Εξεταστική Επιτροπή

Επίκ. Καθ. Αντώνιος Δεληγιαννάκης (ΗΜΜΥ)

Καθ. Μίνως Γαροφαλάκης (ΗΜΜΥ)

Επίκ. Καθ. Βασίλης Σαμολαδάς (ΗΜΜΥ)

Χανιά, Δεκέμβριος 2013

Abstract

During the last decades, data management and storage have become increasingly distributed. Advanced query operators, such as skyline queries, have become, not just useful, but indispensable, in order to help users handle the huge amount of available data by identifying a set of interesting data objects. Space partition techniques, such as recursive division of the data space, have been used for skyline query processing in centralized, parallel and distributed settings. Unfortunately, such grid-based partitioning is not suitable in the case of a parallel skyline query, where all partitions are examined simultaneously, since many data partitions do not contribute to the overall skyline set, resulting in a lot of redundant processing.

This thesis focuses on studying the design, development and comparison of two algorithms to parallel skyline query computation. The presented implementation is based on the Map-Reduce model and its open-source implementation, Hadoop, using space partitioning techniques that were introduced earlier in a parallelizable way over the Map-Reduce model of computation, in one step. A set of experiments will be demonstrated that will show that these techniques are suitable for skyline query processing in parallel architectures. A comparative performance analysis for the two algorithms used, will also be provided. As it will be demonstrated by experimental studies, these techniques are an efficient and scalable solution for skyline query processing in parallel environments.

Περίληψη

Τις τελευταίες δεκαετίες, η διαχείριση και η αποθήκευση των δεδομένων γίνονται όλο και πιο κατανεμημένα. Σύνθετοι τελεστές επερωτήσεων, όπως η επερώτηση κορυφογραμμής, είναι όχι απλώς χρήσιμοι, αλλά απαραίτητοι προκειμένου να βοηθήσουν τους χρήστες να χειριστούν τον τεράστιο όγκο των διαθέσιμων δεδομένων με την αναγνώριση ενός υποσυνόλου από ενδιαφέροντα αντικείμενα. Τεχνικές για τον διαχωρισμό του χώρου των δεδομένων, όπως η αναδρομική κατανομή του χώρου, έχουν χρησιμοποιηθεί για τον υπολογισμό της επερώτησης κορυφογραμμής σε κεντρικοποιημένα, παράλληλα και κατανεμημένα συστήματα. Δυστυχώς τέτοιου είδους διαχωρισμός του χώρου δεδομένων δεν είναι κατάλληλος στην περίπτωση παράλληλων συστημάτων όπου όλα τα χωρίσματα επεξεργάζονται ταυτόχρονα, δεδομένου ότι πολλά από αυτά δεν συμβάλουν στο συνολικό αποτέλεσμα της επερώτησης κορυφογραμμής, με αποτέλεσμα να υπάρχουν πολλοί περιττοί υπολογισμοί.

Η παρούσα διπλωματική εργασία επικεντρώνεται στο σχεδιασμό, την υλοποίηση και την σύγκριση δύο αλγορίθμων για παράλληλο υπολογισμό επερώτησης κορυφογραμμής. Η παρουσιαζόμενη υλοποίησή είναι βασισμένη στο μοντέλο Απεικόνισης - Ελάττωσης και στην ανοιχτού κώδικα υλοποίησή του, το Hadoop. Χρησιμοποιήθηκαν τεχνικές για το διαχωρισμό του χώρου των δεδομένων που έχουν εφαρμοστεί παλαιότερα με τέτοιο τρόπο ώστε να είναι δυνατή η παραλληλοποίησή τους στο σύστημα Απεικόνισης - Ελάττωσης και σε υπολογισμό σε ένα βήμα. Παρατίθεται ένα σύνολο πειραμάτων από τα οποία αποδεικνύεται ότι οι εν λόγω τεχνικές είναι κατάλληλες για παράλληλο υπολογισμό του επερωτήματος κορυφογραμμής. Παρέχεται επίσης μία συγκριτική ανάλυση της απόδοσης των δύο αλγορίθμων που χρησιμοποιούνται. Όπως θα καταδειχθεί από τις πειραματικές μελέτες, οι εν λόγω τεχνικές είναι αποτελεσματικές και δίνουν μία επεκτάσιμη λύση για τον υπολογισμό της επερώτησης κορυφογραμμής σε παράλληλα περιβάλλοντα.

Acknowledgements

First of all, I would like to thank my advisor, Assistant Professor Antonios Deligiannakis, for his support, advising and guidance. Also Dr. Christos Doulkeridis for his help and cooperation during the fulfillment of this thesis.

I would also like to thank Evangelos Vazaios for his suggestions and help during the implementation of the work. Also Xenia Arapi for her availability to help with cluster issues.

Next I would like to thank my family for their support and encouragement, and specially mr. Ioannis Economopoulos for his useful help with this work.

Last but not least I would like to thank my friends, Arodami C., John L., Alexandros M., Vangelis M., John M., Kostas P., Sotiris S., John T..

Stella Maropaki
Chania, December 2013

Contents

List of Figures

LIST OF FIGURES

List of Algorithms

Chapter 1

Introduction

Even before the introduction of skyline queries into database research, the problem of making interesting decisions was known as the maximum vector problem or the Pareto [1] optimum. In recent years, skyline query processing has become an important issue in database research. The popularity of the skyline operator is mainly due to its applicability on decision making applications. In such applications, the database tuples are represented as a set of multidimensional data points and the skyline set contains those particular points which present the best trade-offs between the different dimensions. For example, consider a database that contains information about recreational establishments, such as hotels. Each tuple of the database is represented as a point in a data space consisting of numerous dimensions. In the presented example, the y-dimension represents the price of a room, whereas the x-dimension captures the distance of the hotel to a certain point of interest such as a particular beach. According to the dominance definition, a hotel will dominate another hotel because it will be cheaper and closer to the beach. Thus, the skyline points are the best possible trade-offs between price and distance from the beach.

1.1 Thesis Contribution

The main scope of this thesis is to present an approach to efficiently compute skyline query in a parallel manner using Map-Reduce model in its open-source implementation, Hadoop. In earlier work, space partitioning techniques have been introduced for their use in parallel skyline computation, and this thesis uses them over the Map-Reduce model for computation in one step. Two methods are used in order to compute the skyline query,

one Grid-based and one Angle-based, and a comparative performance analysis is provided for them. For these two methods different space-partitioning techniques is used in order to partition the data so they are parallely processed. The Grid-based method uses grid partitioning technique over the Cartesian coordinates of the data, and the Angle-based method partitions the data using their hyper-spherical coordinates. As demonstrated by the experimental studies the benefits of the proposed methods are numerous, yet the most important one is that the skyline query is computed efficiently, and is more scalable in parallel environments.

1.2 Thesis Outline

Chapter 2 describes the skyline query and space partitioning. Furthermore, it provides basic background information about the Map-Reduce Framework and Hadoop. In Chapter 3, the problem about parallel skyline query is defined and the reason why it is important to be processed in distributed environments. A reference is also made to the previous work published as of the techniques both in parallel and sequential approaches. Chapter 4 describes in detail the approach chosen for both two algorithms, Grid-based and Angle-based Partitioning. In Chapter 5, a presentation on the way of expressing the problem in Map-Reduce Framework is made, including the choices for computing the skyline set in only one step for both algorithms. In Chapter 6, the experimental results are presented, describing the dataset used, and discussing the scalability the techniques have. Finally, in Chapter 7, directions for future work and final conclusions are presented.

Chapter 2

Preliminaries

In this section, the preliminaries of this thesis are presented, such as what is a skyline query and what is partitioning. Furthermore, a brief description of programming models and frameworks utilized in this thesis is provided.

2.1 The Skyline query

2.1.1 Skyline Definitions

Skyline queries have been named after the visualization of city skylines. As shown in Figure 2.1a a city's skyline represents the visual effect to the eye produced by the higher buildings' outline as viewed from an non-obstructed distant point, i.e. from the (opposite) coast or the open sea. As introduced in [2], skyline queries help users to make intelligent decisions over complex data, where numerous and conflicting criteria are considered. Consider the example mentioned in Introduction (Chapter 1) with the database containing information about hotels. Assume that a user is looking for hotels that are as cheap as possible and as close as possible to the beach. In this case it is not obvious whether the user would prefer (*i*) a hotel that is situated very close to the beach but is more expensive than others, or (*ii*) a hotel that is very cheap but farther away from the beach. Furthermore, it is difficult to answer the question of how much cheaper should a hotel be, if it is just a little bit farther away from the beach. The skyline query retrieves all those hotels regarding to which no other hotel exists that is cheaper and closer to the

(a) City (b) Hotel

Figure 2.1: Skylines

beach. Figure 2.1b shows a relevant example. Each point represents a hotel with price per night and distance to the beach as coordinates. The result of the skyline set in this example are hotels a, i, m and k.

In order to define the skyline query, first it is imperative to agree on the definition of *domination*. More formally, given a data space D defined by a set of d dimensions $\{d_1, ..., d_d\}$ and a dataset P on D with cardinality n, a point $p \in P$ can be represented as $p = \{p_1, ..., p_d\}$ where p is a value on dimension d. Without loss of generality, let us assume that the value p_i in any dimension d, is greater or equal to zero ($p_i \geq 0$) and that for all dimensions the minimum values are more preferable. Given these the definition of *domination* would be the following:

Definition 1 *: **(Domination)** A point $p \in P$ is said to* dominate *another point $q \in P$, denoted as $p \prec q$, if (1) on every dimension $d_i \in D$, $p_i \leq q_i$ and (2) on at least one dimension $d_j \in D$, $p_j < q_j$.*

It is now time to agree on the definition of *skyline*.

Definition 2 *: **(Skyline)** The* skyline *is a set of points $SKY(S) \subseteq S$ which are not dominated by any other points. The points in $SKY(S)$ will be called skyline points.*

The notion of skyline queries can be extended to subspaces, where a subspace skyline query only refers to a user-defined subset of attributes. In the running example, the hotel

database may contain various other attributes, such as the number of rooms, the size of the room, and the star rating. Each non-empty subset U of D ($U \subseteq D$) is referred to as a *subspace* of D. The data space D is also referred to as a full space of dataset S. Furthermore, a definition for *subspace skyline* of U could be the following:

Definition 3 *: **(Subspace Skyline)** The* subspace skyline *of U is a set of points $SKY_U(S) \subseteq S$ which are not dominated by any other point on subspace U.*

Consider for example the two-dimensional dataset S depicted in Figure 2.1b. The skyline points are $SKY(S) = \{a, i, m, k\}$, which are the best possible trade-offs between price and distance from the beach. On the other hand, for the subspace $U = \{x\}$ the subspace skyline is $SKY_U(S) = \{a\}$.

Skyline queries have also been studied for the case where constraints exist. Typically, each constraint is expressed as a range along a dimension and the conjunction of all constraints form a hyper-rectangle in the d-dimensional attribute space. Therefore a definition of *constrained skyline* could be the following.

Definition 4 *: **(Constrained Skyline)** The* constrained skyline *returns the skyline set of the subset of the points S' that satisfy the given constraints.*

For example, for a user, a hotel may be interesting only if the price of the room is in the range of $100 to $200. Given this constraint, the skyline set is retrieved from a subset of S that contains all points that satisfy the constraint. In the example of Figure 2.1b, the constrained skyline points are $\{m, i\}$ with respect to the above mentioned constraint on the price.

Finally, literature also proposes *dynamic skyline* queries, a definition of which could be the following.

Definition 5 *: **(Dynamic Skyline)** Given a data space D and a dataset S and m dimension functions $f_1, f_2, ..., f_m$ such that each function $f_i (1 \leq i \leq m)$ takes as parameters a fraction of the coordinates of the data points, the dynamic skyline query returns the skyline set of S according to the new data space with dimensions defined by $f_1, f_2, ..., f_m$*

For example, consider that each hotel in the database is described by its x and y coordinates, and its price. A user may be interested in minimizing the distance to his/her current location (q_1, q_2) in terms of Euclidean distance and the price of the hotel. Thus, each hotel is described in a 2-dimensional space defined by the functions f_1 and f_2 as $f_1(p) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2}$ and $f_2(p) = p3$, where p_1 is the x coordinate, p_2 is the y coordinate and p_3 is the price.

2.1.2 Skyline Computation

Since the introduction of skyline queries [2] in 2001, over a hundred papers have been published with skyline computation in centralized and distributed environments in well-known database conferences or journals. These papers have not only studied efficient skyline computation in centralized or distributed systems but also proposed variations on the traditional skyline operator and studied different premises. Börzsönyi et al. [2] first introduced the skyline operator and presented two basic main memory algorithms: BNL (Block Nested Loops) and D&C (Divide & Conquer). The BNL algorithm uses a block nested loop to compare each tuple of the database with every other tuple. A tuple is reported as a result only if it is not dominated by any other tuple. The D&C algorithm recursively divides the set of input tuples into smaller sets (regions), computes the individual skyline for each region separately, and merges them into the final skyline. SFS (Sort-First-Algorithm) [3] and LESS [4] improve performance of BNL by first sorting tuples according to a monotone function. The main principle of sort-based approaches is that if the tuples are ordered based on a monotone scoring function, then no tuple can be dominated by subsequent tuples.

Skyline query processing with the use of index structures was first proposed by Börzsönyi et al. [2], but elaborated on in later works [5, 6]. The key idea is to use an index to determine dominance between tuples and to prune tuples from further consideration at an early stage. Algorithms using an R-Tree were also proposed, namely NN-search (Nearest Neighbor) [5] and BBS (Branch and Bound Skyline) [6]. These algorithms first compute the nearest neighbor to the origin, which is guaranteed to be part of the skyline result set. Obviously, the region dominated by the nearest neighbor can safely be pruned from consideration. By looking repetitively for the next nearest neighbor in the

non-dominated regions, the complete skyline is determined. It was shown that BBS [6] guarantees minimum I/O costs on a dataset indexed by an R-Tree

2.2 Space Partitioning

Space Partitioning [7] is the process of dividing a space into two or more non-overlapping regions. Any point in the space can then be identified to lie in exactly one of the regions. There are many techniques for partitioning, and most of them seem to be hierarchical, meaning that they recursively divide the data space. Many applications for data processing, especially big data processing, use several partitioning techniques in order to divide their data to be processed in parallel or distributed environments. Some of the most common techniques are:

- *Random Partitioning:* A straight forward approach to partition a dataset among a number of regions is to choose randomly one of the regions for each data.

- *Binary Space Partitioning:* [8] The space is partitioned along a hyperplane into two half-spaces, then either half-space is partitioned recursively until every sub-problem contains only a trivial fraction of the input objects.

- *Quad trees:* Are most often used to partition a two-dimensional space by recursively subdividing it into four quadrants or regions. The regions may be square or rectangular, or may have arbitrary shapes.

- *Grid Space:* It is also often used to partition a two-dimensional space. It is based on recursively dividing the dimensions of the space into two parts. This causes the regions created to have a square or rectangular shape.

- *R-Tree:* This partitioning technique divides the data by referencing them in a tree index. The key idea of the data structure is to group nearby objects and represent them with their minimum bounding rectangle in the next higher level of the tree. At the leaf level, each rectangle describes a single object and at higher levels describes the aggregation of an increasing number of objects.

In this thesis the Hyper-spherical Partitioning technique shall be used. It is based on the hyper-spherical space coordinates. Also the Grid Space Partitioning technique shall be used. Those techniques will be discussed further in Chapter 4.

2.3 Map-Reduce Framework

Map-Reduce is a programming model introduced by Google [9] in 2004, to support distributed computing on large datasets. Today Google's Map-Reduce framework is used inside Google to process data on the order of petabytes on a network of few thousand computers. The framework is inspired by *map* and *reduce* functions, commonly used in functional programming.

Map-Reduce is Turing Complete. This definition describes a system in which a program can be written in order to always find an answer, although without any guarantees regarding runtime or memory [10], so all problems can be expressed in this model. However, it does not provide advantages for all problems. The ones that match its philosophy are the ones that :

- processes parts of data independently from each other.
- require only batch computations (on static data sets).
- work with input data easily expressed as $< key, value >$ pairs.
- handle huge load, even TB's of data.
- can be expressed as a sequence of Map and Reduce functions

For problem cases handling moderate size of input data, the usage of this model is unworthy, since it leads to delays concerning data partitioning and task sharing, that are comparable with execution's runtime.

The basic characteristic of this model is that the whole processing is divided in two basic steps: *map* and *reduce*, while all input/output/intermediate data are being encoded as pairs of a *key* and a *value* attached to this key. Input data have to be organized in such pairs and the framework is responsible for the map function's execution and the output of

intermediate $< key, value >$ pairs. Map-Reduce framework processes this output before transferring it to the reduce phase. This processing sorts and groups the $< key, value >$ pairs by *key*. Finally, the reduce function processes the aforementioned tuples and usually emits a "reduced" set of them.

To better understand the Map-Reduce framework, lets consider an example. Given in the algorithm (1) below are the map and reduce functions for categorizing a set of numbers as even or odd. This is a very simple example where both the Map and the Reduce functions do not extract anything interesting. But as it will be shown in the coming chapters, it is possible to produce something much more complex through these functions.

Algorithm 1 Even or Odd Map-Reduce example

Input: list of numbers

```
1: procedure MAP(String key, Integer value)
2: for each v in values do
3:    if v%2==0 then
4:       emit < "even", v >
5:    else
6:       emit < "odd", v >
7:    end if
8: end for
9: end procedure
10: procedure REDUCE(String key, Iterator value)
11: String val = values.next()
12: while values.hasNext() do
13:    val = val + ", " + values.next()
14: end while
15: emit <key, value>
16: end procedure
```

2.3.1 Map-Reduce model key ideas

Map-Reduce is established as the most popular parallel programming model handling massive data, over the last few years, due to its numerous "big ideas". Below there are some of the most important of them.

- *Scale* out*, not* up*:* For data-intensive workloads, a large number of commodity low-end servers (i.e., the scaling out approach) is preferred over a small number of high-end servers (i.e., the scaling up approach). The latter approach of purchasing symmetric multi-processing (SMP) machines with a large number of processor sockets (dozens, even hundreds) and a large amount of shared memory (hundreds or even thousands of gigabytes) is not cost effective, since the costs of such machines do not scale linearly (i.e., a machine with twice as many processors is often significantly more than twice as expensive). On the other hand, the low-end server market overlaps with the high-volume desktop computing market, which has the effect of keeping prices low due to competition, interchangeable components, and economies of scale.

- *Assume failures are common - providing fault tolerance:* At warehouse scale, failures are not only inevitable, but commonplace. A simple calculation suffices to demonstrate: let us suppose that a cluster is built from reliable machines with a meantime between failures (MTBF) of 1.000 days (about three years). Even with these reliable servers, a 10.000-server cluster would still experience roughly 10 failures a day. For the sake of argument, let us suppose that a MTBF of 10.000 days (about thirty years) were achievable at realistic costs (which is unlikely). Even then, a 10.000-server cluster would still experience one failure daily. This means that any large-scale service that is distributed across a large cluster (either a user-facing application or a computing platform like Map-Reduce) must cope with hardware failures as an intrinsic aspect of its operation. That is, a server may fail at any time, without notice. For example, in large clusters disk failures are common and RAM experiences more errors than one might expect. Data centers suffer from both planned outages (e.g., system maintenance and hardware upgrades) and unexpected outages (e.g., power failure, connectivity loss, etc.). Mature implementations of the Map-Reduce programming model are able to robustly cope with failures through a number of mechanisms such as automatic task restarts on different cluster nodes.

- *Move processing to the data:* In traditional high-performance computing (HPC) applications (e.g., for climate or nuclear simulations), it is commonplace for a supercomputer to have processing nodes and storage nodes linked together by a high-capacity inter-connector. Many data-intensive workloads are not very processor-demanding, which means that the separation of computing and storage creates a bottleneck in the network. As an alternative to moving data around, it is more efficient to move the process around. That is, Map-Reduce assumes an architecture where processors and storage (disk) are co-located. In such a setup, it can be taken advantage of data locality by running code on the processor directly attached to the block of data needed. The distributed file system is responsible for managing the data over which Map-Reduce operates.

- *Process data sequentially and avoid random access:* Data-intensive processing means, by definition, that the relevant datasets are too large to fit in memory and must be held on disk. Seek times for random disk access are fundamentally limited by the mechanical nature of the devices: read heads can only move so fast and platters can only spin so rapidly. As a result, it is desirable to avoid random data access, and instead organize computations so that data is processed sequentially. A simple scenario poignantly illustrates the large performance gap between sequential operations and random seeks: assume a 1 terabyte database containing 10^{10} 100-byte records. Given reasonable assumptions about disk latency and throughput, a back-of-the-envelop calculation will show that updating 1% of the records (by accessing and then mutating each record) will take about a month on a single machine. On the other hand, if one simply reads the entire database and rewrites all the records (mutating those that need updating), the process would finish in less than a work day on a single machine. Sequential data access is, literally, orders of magnitude faster than random data access. The development of solid-state drives is unlikely to change this balance for at least two reasons. First, the cost differential between traditional magnetic disks and solid-state ones remains substantial: large-data will for the most part remain on mechanical drives, at least in the near future. Second, although solid-state disks have substantially faster seek times, order-of-magnitude differences in performance between sequential and random access still remain. Map-Reduce is primarily designed for batch processing over large datasets. To the extent

possible, all computations are organized into long streaming operations that take advantage of the aggregate bandwidth of many disks in a cluster. Many aspects of Map-Reduce's design explicitly trade latency for throughput.

- *Hide system-level details from the application developer:* The challenges in writing distributed software are greatly compounded - the programmer must manage details across several threads, processes, or machines. Of course, the biggest headache in distributed programming is that code runs concurrently in unpredictable orders, accessing data in unpredictable patterns. This gives rise to race conditions, deadlocks, and other well-known problems. Programmers are taught to use low-level devices such as mutes and to apply high-level "design patterns" such as producer-consumer queues to tackle these challenges, but the truth remains: concurrent programs are notoriously difficult to reason about and even harder to debug. Map-Reduce addresses the challenges of distributed programming by providing an abstraction that isolates the developer from system-level details (e.g., locking of data structures, data starvation issues in the processing pipeline, etc.). The programming model specifies simple and well-defined interfaces between a small number of components, and therefore it is easy for the programmer to reason about. Map-Reduce maintains a separation of how computations should be performed and how those computations are actually carried out on a cluster of machines. The first is under the control of the programmer, while the second is exclusively the responsibility of the execution framework or runtime.

2.3.2 Hadoop & HDFS

Today, Hadoop [11] is the most well-known open-source implementation of the Map-Reduce programming model. It has a worldwide impact and many enterprises such as Yahoo!, Last.fm, Facebook and New York Times are using it. It is being implemented in Java and supports multiple classes facilitating code development. Map-Reduce programs executed in Hadoop may be developed in other languages (apart Java) as well, such as Python, Ruby and C++. HDFS is the distributed file system used by Hadoop, sharing many common characteristics with Google's File System(GFS) [12]. Since those characteristics have been analyzed in the Map-Reduce section, reference shall be made to them at this point, from a viewpoint focused on HDFS:

- *High fault tolerance:* In large-scale distributed systems, hardware failures are a commonplace and HDFS cares to locate those nodes and protects the user from losing data and unpredictable crashes.
- *Streaming data access:* HDFS was created to process large scale data in batches (batch processing) and for high throughput achievement.
- *Large input data:* HDFS supports very large file storage by splitting them into blocks.
- *Calculation transfer is cheaper than data transfer:* It is obvious that calculations are more efficient when they are executed close to the data in use. Performance difference is perceptible for large scale input data. HDFS prefers to transfer calculations to other nodes, than transferring respective data, so it possesses mechanisms permitting applications to be moved closer to data - achieving better data locality.

2.3.3 How Map-Reduce works

In order to use the Map-Reduce system, a user must first submit a Job task with the

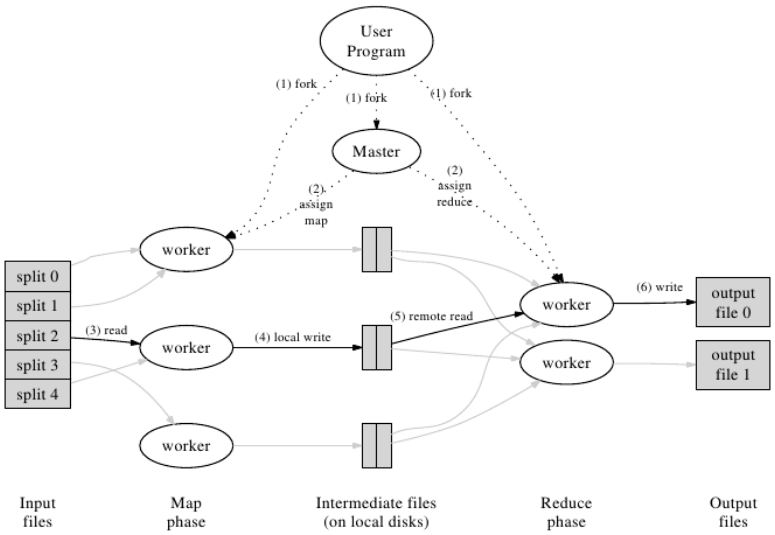


Figure 2.2: Map-Reduce Execution Overview

function-classes made, according to the wanted application. The Job must contain information useful to the execution of the application, such as the number of Map and Reduce tasks that will be used, the location of data input, the location in which the output data will be written, etc.. Figure 2.2 shows the execution overview. After the Job is submitted, the execution begins and many copies of the user program start, where one instance of them becomes the Master. The system tries to utilize data localization by running Map tasks on machines with data while the Master finds idle machines and assigns to them tasks. Depending on the number of Map tasks, the input data are partitioned with a partitioning function, such as hash function, and Map tasks read in contents of corresponding input partition. The Map tasks read the input data and compute $< key, value >$ pairs which are then written to the local disk. After the Map tasks have ended, the Reduce tasks take over iterating over the ordered intermediate Map's output data. The intermediate sorting step is needed in order to group-by the keys that encountered to the Map task, so that the Reduce task will get them as input in the right format (i.e., $< key, [value_1, value_2, ..., value_N] >$). Finally the Reduce task's output is written to output file on global file system and the Master wakes up the user program so that the user will be informed about the output.

There are also some advanced features, other than the basic Map and Reduce, such as:

- *Combiner:* This class starts to run after the Map and before the Reduce tasks. The Combiner is a mini-Reduce task which operates only on data generated by one machine.
- *Input Split:* This class represents the data to be processed by an individual Map.
- *Record Reader:* This class converts byte-oriented view to $< key, value >$ format.

Furthermore it should be indicated that not every problem can be solved with only one Map-Reduce Job. For this reason it is possible that there can be *Job chains*, which means continuous Job tasks one after the other. In this case the first Reduce task's output is the second Map input etc. as shown in Figure 2.3

Figure 2.3: Chain Jobs

Chapter 3

Problem Statement - Related Work

In this section information will be provided about the problem of Parallel Skyline Query and why it is important to be processed in Parallel Environments. Furthermore, a discussion is made on related work about Skyline Query Processing on both parallel and distributed environments.

3.1 Skyline Processing in Parallel Environments

Skyline query is, by its nature, a highly time-consuming process. In a large dataset it can take even a day in order to take out results in a centralized environment. So it is mandatory to find an efficient way to compute not only in centralized, but in distributed and parallel environments too. In this thesis the parallel environment computation is studied.

Assuming a set of N servers S_i participating in the parallel computation. The dataset P is horizontally distributed to the N partitions based on a space partitioning technique, such that P_i is the set of points stored by server S_i where $P_i \subseteq P$, $\bigcup_{1 \leq i \leq N} P_i = P$ and $P_i \cap P_j = \oslash$ for all $i \neq j$.

Observation 1 *: A point $p \in P$ is a skyline point $p \in SKY_P$ if there exists a partition $P_i (1 \leq i \leq N)$ with $p \in P_i \subseteq P$ and $p \in SKY_{P_i}$.*

In other words, the skyline points over a horizontally partitioned dataset are a subset of the union of the skyline points of all partitions. Based on this observation the skyline

points can be computed in two phases, assuming the dataset is horizontally partitioned. In phase one each server S_i computes locally the skyline SKY_{P_i} (also called *local skyline points*) based on the locally stored points P_i. In phase two the local skyline points are merged into a global skyline set by computing the skyline of the local skyline sets. This observation guarantees that the parallel skyline algorithm returns the exact skyline set, after merging the local skyline result sets independently from the partitioning algorithm.

Consider now a parallel architecture where there exists one central server, called *coordinator*, which is responsible for a set of N servers. When a skyline query is submitted, the coordinator distributes the processing task to the N servers. First, the input data is partitioned to the N servers. Then, each server computes the skyline over its local data and returns its local skyline result set to the coordinator. Finally, the coordinator merges the result sets and computes the global skyline result.

It is obvious that the skyline query performance depends on the efficiency of the local skyline computation and the performance of the merging phase. Thus the main objective is to minimize the query execution time. There are several factors that affect the execution time, such:

- *Total processing time:* The total processing time is the aggregation of the execution time each server takes to evaluate the skyline query locally. Therefore, minimizing the local execution time results in minimizing the total processing time. This is accomplished by utilizing an efficient space partitioning method for distributing the dataset among the N servers. Moreover, it is important to exploit parallelism in order to minimize the total processing time. When servers compute their own local skyline simultaneously, then the total processing time will be the local execution time of the server with the longer one.

- *Merging phase time:* The merging phase time is defined by the time it takes for the coordinator to compute the global skyline result set from the local skyline sets of each server. Some factors that may affect this time are the number of the local skyline sets, as well as the number of the servers. Merging phase time is also affected by the number of points each local skyline set has. Thus in order to minimize merging phase time it is needed to minimize the local skyline sets and the

number of points in them. One of the ways this may be accomplished is to make an efficient partitioning of the input data in order to make the local result skyline set smaller. Another way is to experiment on the number of the servers in order to use as less as possible so the merging phase time is minimized.

However, as it will be shown, the factors are often contradictory and there exists a trade-off between them.

3.2 Related Work

As discussed in Section 2.1.2 methods for skyline computation in centralized environments have been examined. Nonetheless lately there is a growing interest in distributed and parallel skyline computation too.

Hose et al. in [13] made a survey of skyline processing in distributed environments that leads to a taxonomy of existing approaches, especially in peer-to-peer systems. Also, in [14], Vlachou et al. use subspace skyline computation over super-peer network systems. In [15], Balke et al. introduce how to efficiently perform distributed skyline queries and thus essentially extend the expressiveness of querying web information systems. They also present useful heuristics to further speed up the retrieval of the skyline query results. In [16], the authors focus on Peer Data Management Systems (PDMS), where each peer provides its own data with its own schema. Huang et al. [17] assume a setting with mobile devices communicating via an ad-hoc network (MANETs), and study skyline queries that involve spatial constraints.

There are also approaches for peer-to-peer skyline computation that apply space partitioning techniques. Wang et al. [18] use the z-curve method to map the multidimensional data space to one dimensional values, that can then be assigned to peers connected in a tree overlay like BATON [19]. Also in [20], the authors use a space partitioning method that is based on an underlying semantic overlay (Semantic Small World - SSW). The main difference between these approaches is that the first uses a tree structure and the second the SSW overlay. However both approaches have the same drawback: some of the peers compute the skyline query without contributing to the result.

The problem of parallel skyline queries over share-nothing architectures was first addressed by Wu et al. in [21]. They proposed DSL algorithm, who relies on space partitioning techniques. With two mechanisms used, recursive region partitioning and dynamic region encoding, the system pipelines participating machines during query execution and minimizes inter-machine communication. In [22], the authors use random partitioning on the dataset to the participating machines and then each machine computes the skyline over its local data using an R-Tree as indexing structure. Another parallel skyline computation is presented in [23] but with different approach. The authors use a multi-disk architecture with one processor and they use the parallel R-Tree in order to access more entries from several disks simultaneously.

Parallel skyline computation is also studied in [24]. As a framework for parallel computation, the authors use both the MP model, which requires that the data is perfectly load-balanced, and a variation of the GMP model, which demands weaker load balancing constraints. In addition to load balancing they minimize the number of blocking steps, where all processors must wait and synchronize. Another approach to increase performance of skyline query processing has been introduced in [25] where the authors use FPGAs for accelerating simple but compute-intensive operations, such as pre-filtering and compression of data.

Finally, the thesis' approach is based on [26] where Vlachou et al. use a novel angle-based partitioning technique using the hyper-spherical coordinates of the data points in order to equally spread skyline points in all partitions so that no machines perform redundant work.

Chapter 4

Approach

In this Chapter the approach for the solution of the problem introduced in Chapter 3 is presented. It is based on two partitioning algorithms, the Grid-Based partitioning, which is described in section 4.1, and the Angle-Based partitioning, which is described in section 4.2. In both approaches, the space partitioning algorithms are used in order to first partition the data space and then using the observation, which was presented in section 3.1, the local skyline of each partition is computed and merged for the global skyline result set.

4.1 Grid-Based partitioning

The most prevalent method for space partitioning regarding skyline query processing is the Grid-based partitioning. As mentioned before, the grid-based partitioning scheme is based on recursively dividing dimensions of the data space. If the data space has two (2) dimensions then the result of the Grid-based partitioning is rectangle-shaped partitions. In order to define the boundaries of the partitions in each dimension the equation $x = \lfloor \sqrt[d]{N} \rfloor$ is used, where x is the number of the boundaries in each dimension, d is the number of dimensions and N is the desired number of partitions.

In this type of partitioning, each server is responsible for a single partition. The main advantage of this approach is that each partition has approximately the same cardinality of data points, therefore the workload of each server is balanced. In addition, each partition has the same distribution of data and computes proportional local skyline points.

However there are several drawbacks in this partitioning technique. First of all, the server corresponding to the origin of the axes contributes the most to the result set, while several others, especially those far away from the axes, do not contribute at all. Furthermore, each server returns, roughly, an equal amount of local skylines, but most of them do not contribute to the global skyline result set. As a result, the merging phase has redundant workload and the communication cost is not minimized.

4.2 Angle-Based partitioning

As introduced in [26], Angle-based partitioning maps cartesian coordinate space into hyper-spherical space. Then the data space is partitioned into N partitions on the angular coordinates. Hyper-spherical coordinates consist of a radial coordinate r and $d-1$ angular coordinates $\phi_1, \phi_2, ..., \phi_{d-1}$, where d is the cartesian space dimension. These coordinates are computed from the following equations:

$$r = \sqrt{(x_n)^2 + (x_{n-1})^2 + \cdots + (x_1)^2} \tag{4.1}$$

$$\tan\phi_1 = \frac{\sqrt{(x_n)^2 + (x_{n-1})^2 + \cdots + (x_2)^2}}{x_1}$$

$$\cdots$$

$$\tan\phi_{d-2} = \frac{\sqrt{(x_n)^2 + (x_{n-1})^2}}{x_{n-2}}$$

$$\tan\phi_{d-1} = \frac{x_n}{x_{n-1}}$$

Notice that generally $0 \leq \phi_i \leq \pi$ and $0 \leq \phi_{d-1} \leq 2\pi$, for $i < d-1$, but in this case $0 \leq \phi_i \leq \frac{\pi}{2}$, for $i \leq d-1$. This is because assuming without loss of generality that for any data point x the coordinates x_i are greater or equal to zero ($x_i \geq 0\ \forall i$) for all dimensions.

After mapping the cartesian space into hyper-spherical, the angular coordinates ϕ_i are used to divide the space into N partitions. Specifically, a grid-based partitioning technique is applied over the $d-1$ space defined by the angular coordinates. This leads to a partitioning where all points that have similar angular coordinates fall in the same partition independently from the radial coordinate, i.e. how far the point is from the origin.

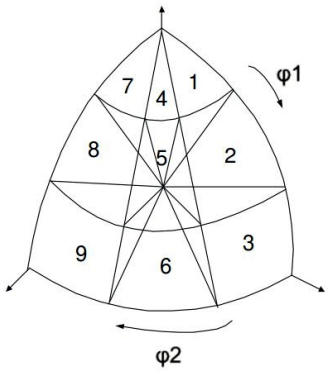


Figure 4.1: Example of 3-dimensional angle-based partitioning

Consider for example a 3-dimensional space. As depicted in figure 4.1 the space is divided in $N = 9$ partitions using the angular coordinates ϕ_1 and ϕ_2. Imagine a theoretical scenario where there are infinite number of servers available. Then using this partitioning scheme, each partition is reduced to a line, as all points in the same partition have similar angular coordinates. It is well-known that correlated dataset are data distributed around a line starting from the origin of the space, and have skyline sets of small cardinality. Thus in this theoretical scenario each server would be assigned with a correlated data set. Therefore, by increasing the number of partitions, the performance and the skyline cardinality similar to that of a correlated dataset is achieved in every partition, even if the overall distribution of the dataset is not correlated.

Given the number of partitions N and a d-dimensional data space D, the angle-based partitioning assigns to each partition a part of the data space D_i $(1 \leq i \leq N)$. The data space of the i^{th} partition is defined as: $D_i = [\phi_1^{i-1}, \phi_1^i] \times ... \times [\phi_{d-1}^{i-1}, \phi_{d-1}^i]$, where $\phi_j^0 = 0$ and $\phi_j^N = \frac{\pi}{2} (1 \leq j \leq d)$, while ϕ_j^{i-1} and ϕ_j^i are the boundaries on the angular coordinate ϕ_j for the partition i. After assigning to each partition a part of the data space D_i, the distribution of the data points to these partitions can easily be made. The $d-1$ angular coordinates of every point p are compared with the boundaries of each partition and the corresponding partition is found.

The intuition of this partitioning scheme is that all partitions share the area close to the origin of the axes. This is important because it increases the probability that the

global skyline points which exist near the origin of the axes are distributed evenly among the partitions. As mentioned, the angle-based partitioning is expected to return only a few local skylines. Therefore, achieving a small local result set, that leads to smaller network communication costs and smaller processing costs for the merging phase.

Chapter 5

Implementation

The implementation will be described here. Information about the Structure of the code, the tools and the programming language used, will be provided. Also it will be shown how the approach is implemented in a Map-Reduce System.

5.1 Structure

This implementation, based on the two partitioning algorithms mentioned before in Chapter 4, was written in programming language *java* and with Eclipse IDE development tool. In order to use the Hadoop's Map-Reduce framework, the package *org.apache.hadoop.mapreduce.** was imported in the code.

As discussed in Chapter 4, the two partitioning techniques was used to partition the dataset. The Job class was used to compute the boundaries of the partitions according to the corresponding technique and then the Map classes was used to map the data points to the partitions. Then the local skyline points was computed with the Combiner class and then merged them into the Reduce class to achieve the global skyline result set. Each class is analyzed in the following sections.

5.2 Job Class

This is the start class where the input dataset, the number of the desired Reduce classes, the type of the partitioning technique, the desired number of the partitions to be created

and the path of the output dataset are provided. In this class the boundaries of the chosen partitioning technique are computed by reading the first 10% of the data input, and the Job task initializes. According to the chosen partitioning technique, it sets the corresponding Map class. When the job task ends it outputs the execution time.

Specifically, the Algorithm 2 is used to estimate how many boundaries each dimension should have. The dimensions are d for Grid-based partitioning and $d-1$ for Angle-based.

Algorithm 2 Estimated number of boundaries

Input: d=dimensions, N=number of partitions

Output: b[ ]= a list with number of boundaries in each dimension

1: $x = \lfloor \sqrt[d]{N} \rfloor$
2: $b[\,] \leftarrow x$ for all d elements
3: **while** power of $b[\,] \leq N$ **do**
4: // some element i where $0 \leq i < d$
5: $b[i]++$
6: **end while**
7: **return** $b[\,]$

After computing the number of boundaries, they are computed using the Algorithm 3 for Grid-based partitioning.

Algorithm 3 Grid-based partitioning

Input: d=dimensions, b[ ]=the number of boundaries in each dimension, m=max value of dataset

Output: boundaries[ ]= a list with the boundaries in each dimension

1: **for all** $0 \leq i < d$ in dimensions **do**
2: boundaries[i]$\leftarrow m/b[i]$
3: **end for**
4: **return** boundaries[ ]

For computing the Angle-based partitioning boundaries the hyper-spherical coordinates

of the points must first be computed. Using the Algorithm 4, the hyper-spherical coordinates are computed and then the boundaries for the partitions. After computing the

Algorithm 4 Angle-based partitioning

Input: d=dimensions, b[ ]=the number of boundaries in each dimension,
points[ ]=list with the Cartesian coordinates of the points
Output: boundaries[ ]= a list with the boundaries in each dimension

1: **procedure** COMPUTE_R(point x)
2: **for each** dimension d **do**
3: $r = \sqrt{(x_d)^2 + (x_{d-1})^2 + \cdots + (x_1)^2}$
4: **end for**
5: **return** r
6: **end procedure**
7:
8: **procedure** COMPUTE_Φ(String key, Iterator value)
9: **for each** dimension $0 \leq i < d - 1$ **do**
10: $\phi_i = (\sqrt{(x_d)^2 + (x_{d-1})^2 + \cdots + (x_{i+1})^2})/x_i$
11: **end for**
12: **return** all ϕ
13: **end procedure**
14:
15: **procedure** ANGLE_PARTITIONING( )
16: **for all** points x **do**
17: list_ϕ.add(COMPUTE_Φ(x))
18: list_r.add(COMPUTE_R(x))
19: **end for**
20: sort(list_ϕ)
21: max ← list_ϕ.maxValue()
22: min ← list_ϕ.minValue()
23: **for all** $0 \leq i < d$ in dimensions **do**
24: boundaries[i]← $(max - min)/b[i]$
25: **end for**
26: **return** boundaries[ ]
27: **end procedure**

boundaries, they are set in the Configuration so that the Map class can accept them. Then the job task is initiated by setting the Map, Combiner and Reduce classes, the number of Reduce class tasks, the input and output $< key, value >$ classes and the input and output format.

5.3 Mapper Class

There are two Map classes, one for each partitioning technique. They both assign every point to its partition according to the partition boundaries computed before in the Job class. If the partitioning technique is Grid-based, then the assignment is based on cartesian coordinates of every point.If the technique is Angle-based, the assignment is based on the angular coordinates. Given below is the assignment Algorithm 5. The Map class

Algorithm 5 Partition assignment

Input: d=dimensions, b[ ]=the boundaries in each dimension, p=the point read
Output: $< partition\_name, point >$ pairs

```
1: for each boundary in b[i] do
2:    if point ∈ b[i] then
3:       emit < partition_name, point >
4:       exit
5:    else
6:       i++
7:    end if
8: end for
```

emits $< key, value >$ data so in this case *key* is the name of the partition and *value* in the point assigned to the partition.

5.4 Combiner Class

The Combiner class runs on the output of the Map phase and is usually used as an intermediate step to lessen the keys that are being transferred to the Reducer. In this case, the Combiner class computes the local skyline set of each partition, without taking in account the partitioning technique. The Skyline computation Algorithm 6 used, is based

on Block Nested Loop, discussed in Section 2.1.2 from [2], but with some differences, for optimization reasons.

In this algorithm one list is used for keeping the local skyline points, and a temporary point with the maximum cartesian coordinates of the local skyline points found so far. The main idea for the maximum coordinates is that if a point is dominated by them then it is bound that it will be dominated by at least one point found in the local skyline set. When a point is read it is first checked for dominance with the maximum coordinates. If it isn't dominated, then it is checked for domination with every point in the local skyline list. For the points in the list there are three cases:

1. The point read dominates a point in the local skyline list. Then the point is removed from the list, since it doesn't follow the definition of skyline.

2. The point read is dominated by a point in the local skyline list. Then the process of domination checking with the points in the list stops, and continues with the next point read.

3. The point read does not dominate or is not dominated by any point in the local skyline list. If such a case the point read is added in the list.

If any point from the list is removed or added then a boolean variable is set as "true" and the maximum coordinates are changed to fit the new points. The process stops when all points of the given partition are read from the mapping output and the local skyline set list is then emitted to the Reducer, with all points corresponded to the same key.

5.5 Reducer Class

The Reducer class runs on the output of Combiner. It takes an iterator with values on the same key, $< key, [value_1, value_2, ..., value_N] >$, and merges them into the final output. In this case, the Reducer takes all local skyline sets from the Combiner, since all local skyline points have the same key. It runs the same Algorithm 6 used for computing the Skyline, in the Combiner. Finally the result is emitted to the output file declared in the beginning of the Job.

Algorithm 6 Skyline computation

Input: d=dimensions, p[ ]=the points read in a partition

Output: skyline[ ]= a list with the local skyline points of a partition

```
1: boolean change = true
2: int max_coordinates[ ]
3: skyline.add(p.first)
4: fix max_coordinates
5: change = false
6: int temp = 0
7: for each point p[i]∈p do
8:    if max_coordinates dominate p[i] then
9:       next point
10:   else
11:      for each point s[j]∈skyline do
12:         if p[i] dominates s[j] then
13:            skyline.remove(s[j])
14:            temp++
15:            change = true
16:         else if s[j] dominates p[i] then
17:            end for
18:         else
19:            temp++
20:         end if
21:      end for
22:      if temp == skyline.length then
23:         skyline.add(p[i])
24:         change = true
25:      end if
26:   end if
27:   if change == true then
28:      fix max_coordinates
29:   end if
30: end for
31: return skyline list
```

Chapter 6

Experimental Results

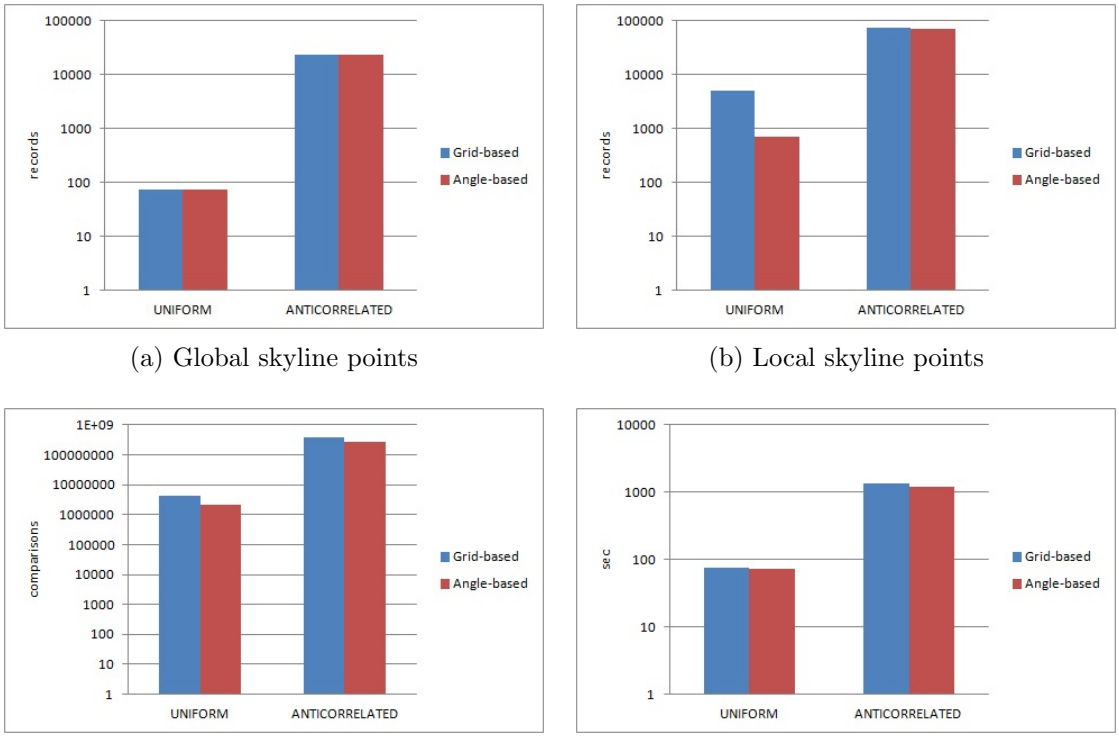
In this chapter the results of this work are presented and it will be shown how it benefits the Skyline Query Processing. The best approach to show that certain techniques, like the ones proposed in Chapters 4 and 5, are effective, would be to show how these techniques are scalable with the distribution of data (Section 6.2), the cardinality (Section 6.3), the dimensionality (Section 6.4) and the number of partitions (Section 6.5).

6.1 Experiments Description

The experiments were performed on cluster clu26.softnet.tuc.gr which had 1 master and 17 slave server nodes, and run the Apache Hadoop release 1.0.3. Its heap size was 888.94 MB and had a 7.43 TB HDFS. It had 68 Map tasks and 68 Reduce task capacity and it assigned in average 8 tasks per node. In order to run the experiments, uniform and anti-correlated datasets were generated with d=2, 3, 4 and 5 dimensions. The points generated were unique in each dataset and had values from 0 to 10.000. The cardinality of each set was n=1.000.000, 5.000.000, 10.000.000, 15.000.000 and 20.000.000 points. There were experiments with both Grid-based and Angle-based partitioning techniques for N=10, 20, 30, 40 and 50 partitions.

6.2 Scenario I: *Data Distribution*

The first step in this evaluation was to compare the two partitioning algorithms and try to calculate the actual difference in computational overhead in terms of data distribution.



(a) Global skyline points

(b) Local skyline points

(c) Comparisons

(d) Execution time

Figure 6.1: Data Distribution for d=3, n=1M, N=30

To this end, the execution time of each algorithm was measured for producing the skyline result set in two types of distributions, uniform and anti-correlated. Figure 6.1 depicts the results in logarithmic scale. In Figure 6.1a it is shown that the result of the global skyline set is the same for the two algorithms, in both uniform and anti-correlated distributed data. Also uniform global skyline points are by far less than the anti-correlated global ones due to the nature of uniform and anti-correlated data distribution.

Figure 6.1c shows the number of comparisons needed for the local skyline computation in terms of candidates examined for domination. As it was expected, the uniform dataset has less comparisons than the anti-correlated one. Also the Grid-based partitioning technique requires more objects to be examined for domination. This is also confirmed by Figure 6.1b where the local skyline points that have been computed in all Combiners are depicted. It is shown that for both uniform and anti-correlated data distribution,

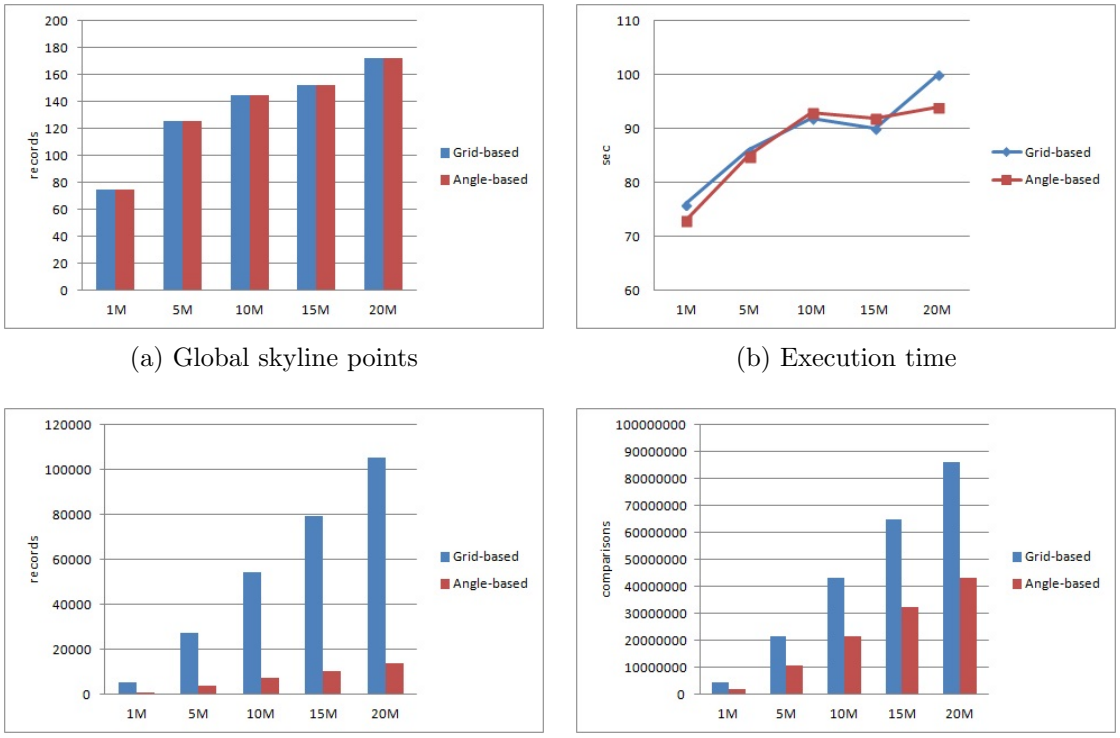
the Angle-based partitioning technique computes less local skyline points than the Grid-based one, and this improves the merging time and, accordingly, the total execution time. Moreover, the local skyline points that do not contribute to the global result are more in the Grid-based partitioning technique than in the Angle-based one.

In addition, although the Angle-based partitioning technique produces less local skyline points, it takes time to compute the boundaries of every partition and to compute the hyper-spherical coordinates of the points. On the other hand, the Grid-based partition does not need much time to compute the boundaries of partitions and the hyper-spherical coordinates of the points, but it computes more local skyline points and so it takes more time to merge them. So the execution time of both algorithms is approximately the same, as shown in Figure 6.1d, for both uniform and anti-correlated data.

6.3 Scenario II: *Scalability with Cardinality*

The next step was to evaluate the performance of the two algorithms on dataset with different cardinality. For this purpose they were used datasets with 3 dimensions and cardinalities of, respectively, 1M, 5M, 10M, 15M and 20M. The distribution of the dataset was both uniform and anti-correlated and the number of partitions was defined as $N = 30$. In Figure 6.2 it is depicted the global skyline points, the execution time, the local skylines points and the number of comparisons needed for them to be produced for uniform dataset, for the two algorithms compared with cardinality of the dataset and in Figure 6.3 they are compared the ones for anti-correlated dataset for this evaluation step.

In Figures 6.2a and 6.3a it is noted that the global skyline result set is the same for both algorithms in both partitioning techniques. It is also observed that the size of the global skyline result set is increasing according to the increase of the cardinality of the input data. Figures 6.2b and 6.3b depict the execution time of each technique. It is shown an increase of the execution time according to the cardinality of the input data. In case of uniform datasets the Angle-based technique has a more stable performance as the cardinality changes over 10M unlike the Grid-based one. This doesn't happen with the anti-correlated dataset due to the nature of anti-correlated distribution. However,



(a) Global skyline points

(b) Execution time

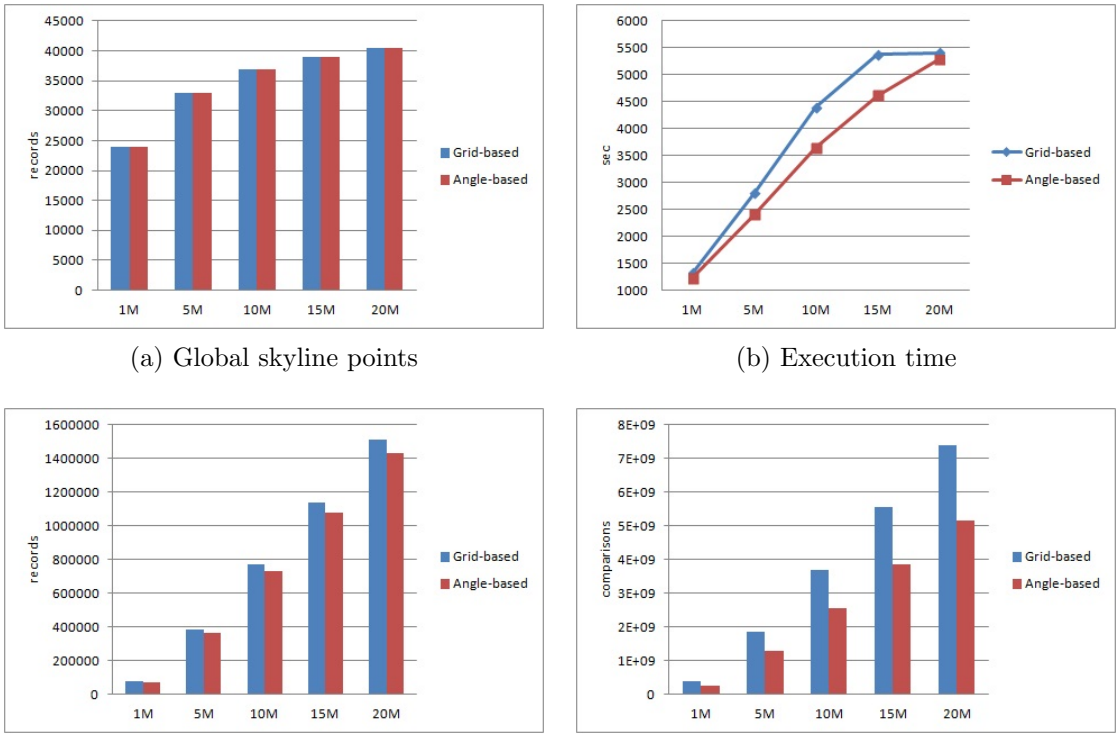
(c) Local skyline points

(d) Comparisons

Figure 6.2: Scalability with cardinality for d=3, N=30, uniform

in both cases of uniform and anti-correlated datasets, the Angle-based technique outperforms Grid-based.

Figures 6.2c and 6.3c show the local skyline points for uniform and anti-correlated dataset produced by the Combiners. It is shown that, as expected, the Grid-based technique produces more local points and many of them do not contribute to the global skyline set. This is also confirmed by Figures 6.2d and 6.3d, where the number of comparisons needed for the local skyline computation is depicted. For both uniform and anti-correlated data the Angle-based partitioning technique requires less comparisons in terms of domination checks, than the Grid-based one. Also it is observed that, as expected, local skylines and comparisons are increased too, according to the cardinality increase for both uniform and anti-correlated dataset.



(a) Global skyline points

(b) Execution time

(c) Local skyline points

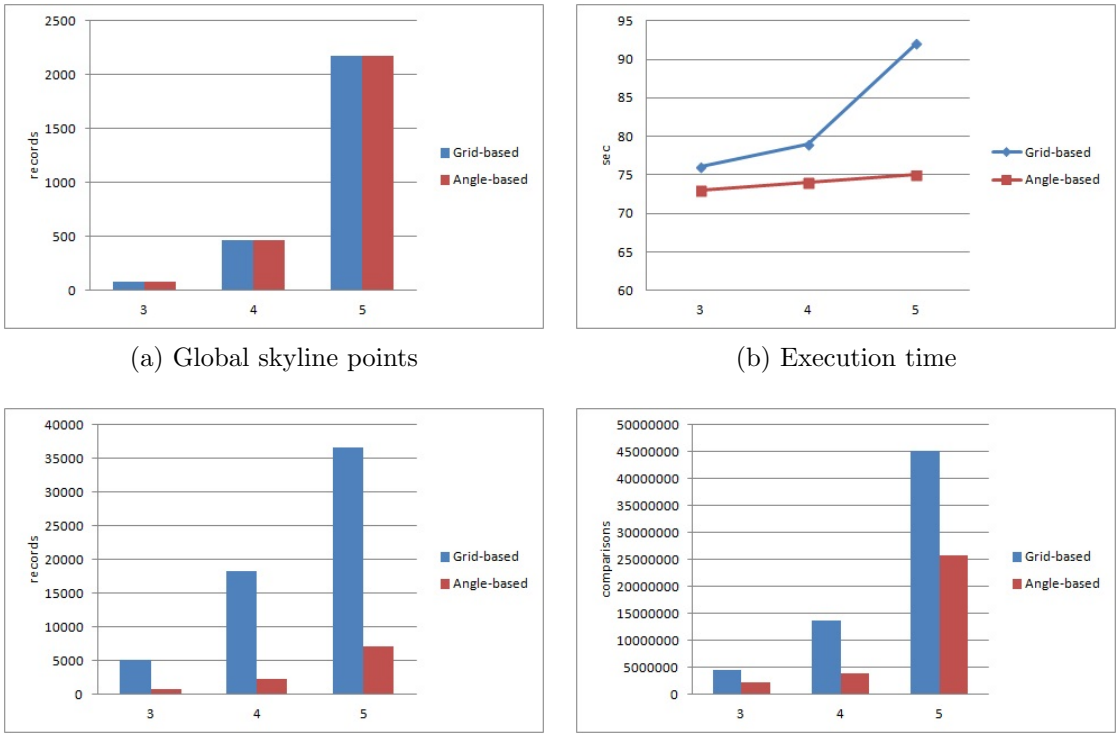
(d) Comparisons

Figure 6.3: Scalability with cardinality for d=3, N=30, anti-correlated

6.4 Scenario III: *Scalability with Dimensionality*

In this step it is evaluated how the two algorithms can be scaled in terms of dimensionality. As previously mentioned, the partitions are defined with $N = 30$ and there were used both uniform and anti-correlated distributed data. The datasets had dimensions from 3 to 5 for uniform and from 2 to 4 for anti-correlated data, with a cardinality of 1M. In Figure 6.4 are depicted the results for a uniform dataset and in Figure 6.5 the ones for an anti-correlated.

In Figure 6.4a and 6.5a it is shown that, as expected, global skyline points are the same for both techniques, and as the number of dimensions vary the size of the result set increases rapidly. The execution time is depicted in Figure 6.4b and 6.5b where it is shown that the Angle-based technique outperforms the Grid-based one, especially for data with 4 and 5 dimensions. In case of unirofm dataset, the Angle-based technique shows a more stable



(a) Global skyline points

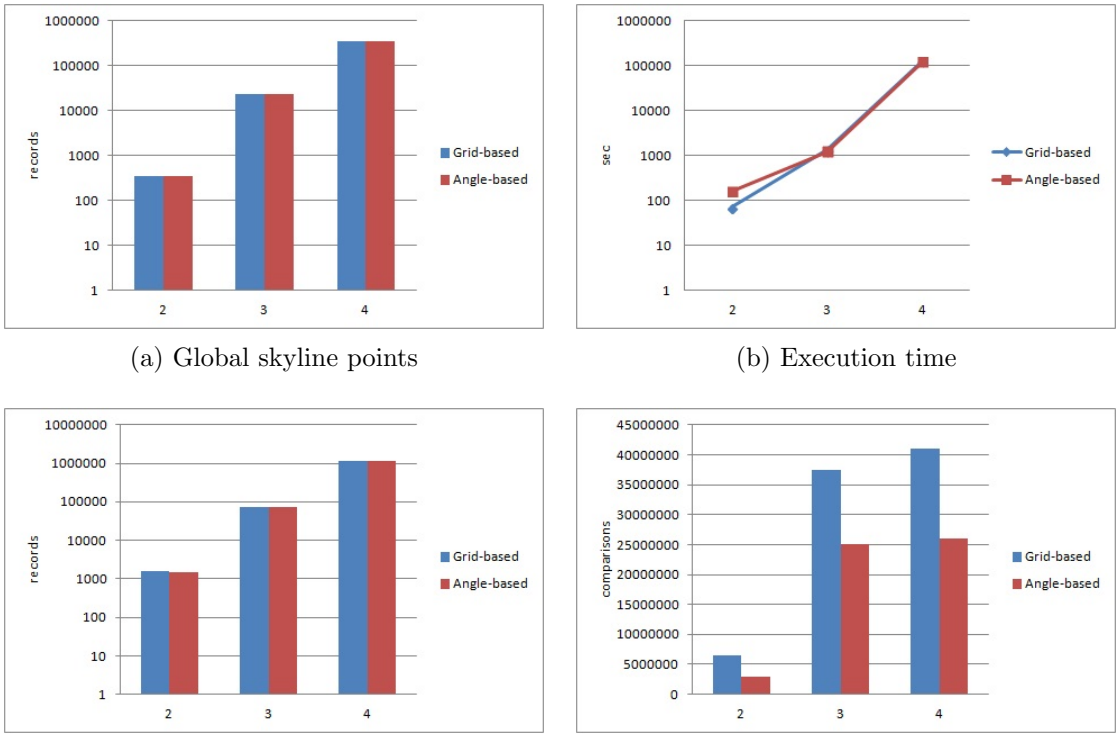
(b) Execution time

(c) Local skyline points

(d) Comparisons

Figure 6.4: Scalability with dimensionality for N=30, n=1M, uniform

performance, in contrast with the Grid-based one. In case of anti-correlated dataset, the growth is in orders of magnitude, due to the nature of anti-correlated dataset. This is also confirmed by the number of local skyline points, shown in Figure 6.4c and 6.5c, and the number of comparisons, shown in Figure 6.4d and 6.5d, where the Angle-based technique also outperforms the Grid-based one. As shown in figures, as the number of dimensions is increased, local skylines and comparisons are increased too. With the Grid-based technique the growth is more rapid, than with the Angle-based one, which is more stable. Also in case of anti-correlated dataset, the increase id more rapid, than in case of uniform one.



(a) Global skyline points

(b) Execution time

(c) Local skyline points

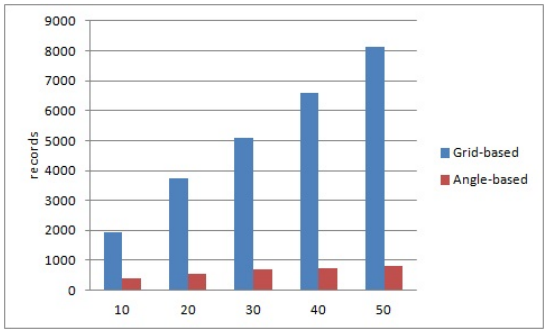
(d) Comparisons

Figure 6.5: Scalability with dimensionality for N=30, n=1M, anti-correlated

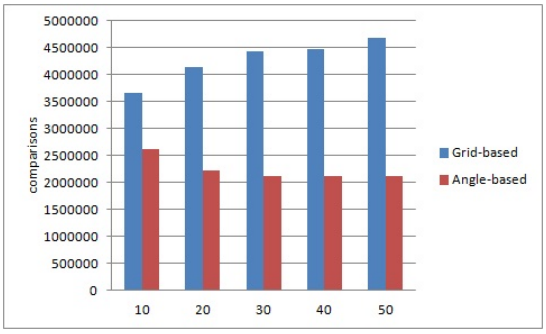
6.5 Scenario IV: *Number of Partitions*

Finally the last experiment regards the way the numbers of partitions affect the performance of the two algorithms. For this experiment both uniform and anti-correlated data were used with 3 dimensions and a cardinality of 1M. The number of partitions was scaled from 10 to 50 with a step of 10. In Figure 6.6 the results are shown for uniform dataset and in Figure 6.7 the ones for anti-correlated dataset. The global skyline results are not depicted because they stay the same for all numbers of partitions. In case of uniform dataset they are 75 and in case of anti-correlated they are 23990.

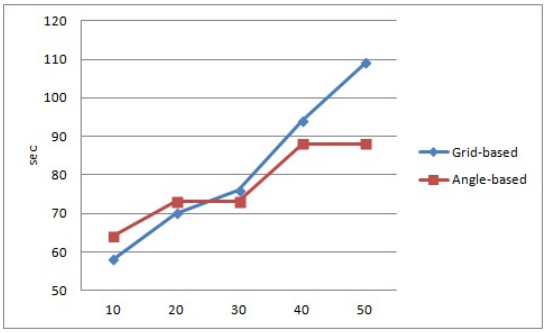
It is shown that for both uniform and anti-correlated dataset, Angle-based technique outperforms Grid-based one, in all terms, execution time, local skyline points, and comparisons. In case of uniform dataset, as depicted in the Figure 6.6c, the execution time increases as the number of partitions increase. In contrary, in case of anti-correlated



(a) Local skyline points



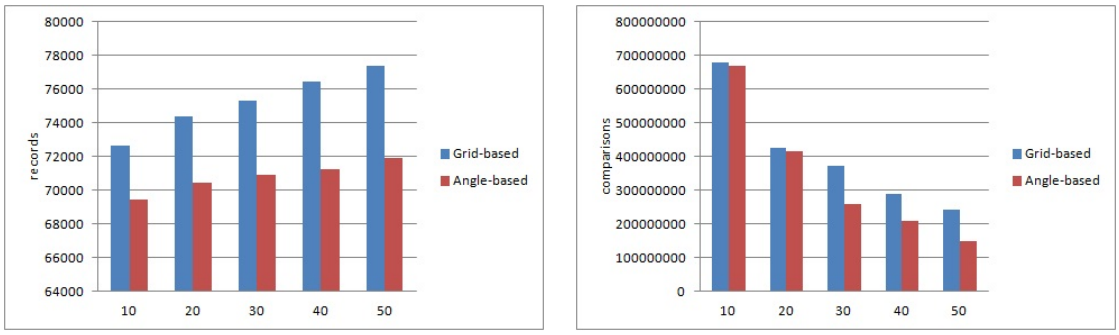
(b) Comparisons



(c) Execution time

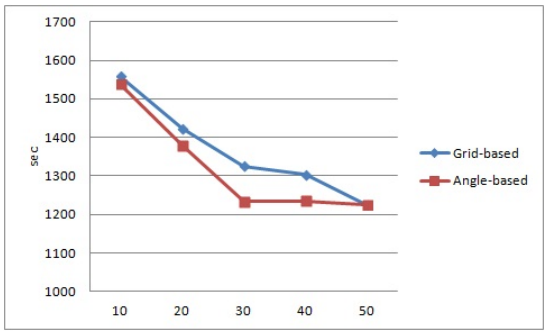
Figure 6.6: Number of partitions for d=3, n=1M, uniform

dataset, as shown in the Figure 6.7c, the execution time reduces. This is due to the fact that as the number of partitions increases, there are more skyline points, as Figures 6.6a and 6.7a show, but in each partition there are less local points to be processed and so less comparisons made, as Figures 6.6b and 6.7b show.



(a) Local skyline points

(b) Comparisons



(c) Execution time

Figure 6.7: Number of partitions for d=3, n=1M, anti-correlated

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis summarizes an approach to efficient skyline query computing in a parallel manner using the Map-Reduce model in its open-source implementation, Hadoop. As explained in Chapter 4, this method alleviates most of the problems of the traditional techniques, thus managing to reduce the response time and fairly share the computation workload. As demonstrated by the experimental results in Chapter 6, this work proves useful in skyline query processing in parallel environments, in term of efficiency and response time. this approach also creates room for further development which will be, in its main points, discussed below.

7.2 Future Work

In future work, the aim is to examine more space-partitioning technique, other than the ones used in this thesis. It is also possible to experiment with the number of nodes running the computation, as well as the number of Map and Reduce tasks. In addition, more dataset could be examined in terms of data distribution and data types.

Another interesting scenario would be to examine the performance over sub-space skyline queries. The Angle-based partitioning technique is not affected by the projection of the data points, during sub-space skyline queries, since again the region near the origin is

equally spread to all partitions. On the other hand, the Grid-based partitioning technique does not use the region near the origin in all partitions, so it may affect the sub-space skyline query computation according to the projection of the sub-space.

Furthermore, in this approach, some kind of sorting in the data in partitions could be added, so that the processing cost of the domination checks is alleviated, and more points are pruned. If this sorting results efficient, the merging phase in the end of the process would be even faster and less needy for the Reduce task.

Finally, it would be interesting to test this, methods, as well as more partitioning techniques, with stream data, as it would be a challenge to compute skyline sets without knowing the hole dataset from the beginning.

References

[1] Wikipedia: Pareto efficiency — wikipedia, the free encyclopedia (2013) http://en.wikipedia.org/wiki/Pareto_efficiency. 1

[2] Börzsönyi, S., Kossmann, D., Stocker, K.: The skylnie operator. In: Proceedings of International Conference on Data Engineering (ICDE). (2001) 3, 6, 29

[3] Chomicki, J., Godfrey, P., Gryz, J., Liang, D.: Skyline with presorting. In: Proceedings of International Conference on Data Engineering (ICDE). (2003) 6

[4] Godfrey, P., Shipley, R., Gryz, J.: Maximal vector computation in large data sets. In: Proceedings of International Conference on Very Large Data Bases (VLDB). (2005) 6

[5] Kossmann, D., Ramsak, F., Rost, S.: Shooting stars in the sky: An online algorithm for skyline queries. In: Proceedings of International Conference on Very Large Data Bases (VLDB). (2002) 6

[6] Papadias, D., Tao, Y., Fu, G., Seeger, B.: An optimal and progressive algorithm for skyline queries. In: Proceedings of International Conference on Management of Data (SIG-MOD). (2003) 6, 7

[7] Wikipedia: Space partitioning — wikipedia, the free encyclopedia (2013) http://en.wikipedia.org/wiki/Space_partitioning. 7

[8] Tóth, C.D.: Binary space partitions: Recent developments (2005) 7

[9] Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: 6th Symposium on Operating Systems Design and Implementation (OSDI '04), Google Inc. (2004) 8

[10] Wikipedia: Turing completeness — wikipedia, the free encyclopedia (2013) http://en.wikipedia.org/wiki/Turing_completeness. 8

[11] The-Apache-Software-Foundation: Apache hadoop (2013) http://hadoop.apache.org/. 12

[12] White, T.: Hadoop: The Definitive Guide. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472 (2009) 12

[13] Hose, K., Vlachou, A.: A survey of skyline processing in highly distributed environments. VLDB Journal **21**(3) (June 2012) 19

[14] Vlachou, A., Doulkeridis, C., Kotidis, Y., Vazirgiannis, M.: Skypeer: Efficient subspace skyline computation over distributed data. In: Proceedings of Conference on Data Engineering (ICDE). (2007) 19

[15] Balke, W.T., Güntzer, U., Zheng, J.X.: Efficient distributed skylining for web information systems. In: Proceedings of International Conference on Extending Database Technology (EDBT). (2004) 19

[16] Hose, K., Lemke, C., Sattler, K.U.: Processing relaxed skylines in pdms using distributed data summaries. In: Proceedings of Conference on Information and Knowledge Management (CIKM). (2006) 19

[17] Huang, Z., Jensen, C.S., Lu, H., Ooi, B.C.: Skyline queries against mobile lightweight devices in manets. In: Proceedings of International Conference on Data Engineering (ICDE). (2006) 19

[18] Wang, S., Ooi, B.C., Tung, A.K.H., Xu, L.: Efficient skyline query processing on peer-to-peer networks. In: Proceedings of International Conference on Data Engineering (ICDE). (2007) 19

[19] Jagadish, H., Ooi, B.C., Vu, Q.H.: Baton: A balanced tree structure for peer-to-peer networks. In: Proceedings of International Conference on Very Large Data Bases (VLDB). (2005) 19

[20] Li, H., Tan, Q., Lee, W.C.: Efficient progressive processing of skyline queries in peer-to-peer networks. In: Proceedings of International Conference on InfoScale. (2006) 19

[21] Wu, P., Zhang, C., Feng, Y., Zhao, B.Y., Agrawal, D., Abbadi, A.E.: Parallelizing skyline queries for scalable distribution. In: Proceedings of Conference on Extending Database Technology (EDBT). (2006) 20

[22] Cosgaya-Lozano, A., Rau-Chaplin, A., Zeh, N.: Parallel computation of skyline queries. In: Proceedings of International Symposium on High Performance Computing Systems and Applications. (2007) 20

[23] Gao, Y., Chen, G., Chen, L., Chen, C.: Parallelizing progressive computation for skyline queries in multi-disk environments. In: Proceedings of International Conference of Database and Expert Systems Applications (DEXA). (2006) 20

[24] Afrati, F.N., Koutris, P., Suciu, D., Ullman, J.D.: Parallel skyline queries. In: Proceedings of International Conference on Database Theory (ICDT). (2012) 20

[25] Woods, L., Alonso, G., Teubner, J.: Parallel computation of skyline queries. In: IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines. (2013) 20

[26] Vlachou, A., Doulkeridis, C., Kotidis, Y.: Angle-based space partitioning for efficient parallel skyline computation. In: Proceedings of International Conference on Management of Data (SIGMOD). (2008) 20, 22