INTERNATIONAL
HELLENIC
UNIVERSITY

Distributed Algorithms for Skyline Computation using Apache Spark

Papanikolaou Ioanna

SID: 3308170016

SCHOOL OF SCIENCE & TECHNOLOGY

A thesis submitted for the degree of

Master of Science (MSc) in Data Science

DECEMBER 2018

THESSALONIKI – GREECE

INTERNATIONAL HELLENIC UNIVERSITY

Distributed Algorithms for Skyline Computation using Apache Spark

Papanikolaou Ioanna

SID: 3308170016

Supervisor:	Prof. Apostolos Papadopoulos
Supervising Committee Members:	Assoc. Prof. Name Surname
	Assist. Prof. Name Surname

SCHOOL OF SCIENCE & TECHNOLOGY

A thesis submitted for the degree of

Master of Science (MSc) in Data Science

DECEMBER 2018

THESSALONIKI – GREECE

Abstract

This dissertation was written as a part of the MSc in Data Science at the International Hellenic University. During the last decade, database technologies have been studied extensively to comply to the Big Data era and the demands for results that lead to complex, decision making processes. The implementation of an efficient skyline computation algorithm has gained a lot of attention because it offers interesting results from multi-criteria queries. In this thesis, different skyline algorithms are implemented in the Apache Spark platform and their performance in a distributed environment is evaluated and discussed.

Ioanna Papanikolaou

07/12/2018

Contents

1 Introduction

Today, after years of radical technological evolution data are everywhere. They come at large volumes, in many forms and are easily accessible. Individuals and organizations are challenged to adjust to this fact and change their process of decision making. A company's manager is now able to base their decision regarding the location of a potentially new store on collected information rather than their personal instinct. A consumer that wishes to buy a new device, is more equipped to take the right decision when they rely on the actual data rather than the seller's suggestions. On the other hand, it is not always the case that a decision can be made by simply processing data and retrieving the optimal suggestion. A very common challenge is to process the data in a way that yields a wide range of useful suggestions the user can review and base their decisions on.

Skyline vectors were introduced in 2001 as a way of satisfying this need and are still widely discussed in scientific literature.

1.1 The skyline problem

The skyline as a term was introduced by *Borzsonyi* [1] who issued the maximal vector computation problem in database applications. Given a set of tuples having an ordering relation on each dimension, Skyline is a subset of all the tuples that are not dominated by any other tuple of the original set. A tuple a dominates another tuple b $(a \prec b)$ when the values of each of a's attributes are bigger than or equal to the corresponding values of b. A real-world example of Skyline exists in an online mobile-phone store. A user is browsing the database aiming to find the best results for a non-expensive phone with an adequately large screen size. These two characteristics are most probably highly anti-correlated, therefore if the results of the user's search query were ordered based on the prices of the available phones, the user would have to ignore many of the top results due to their small screen size and vice versa. A Skyline query, set to maximize the screen attribute and minimize the price, would return to the user a variety of phone options with the property that no other phone exists having the same or smaller price and the same or bigger screen at the same time. The user could then examine their options and decide their preference between cheaper phones with smaller screens or more expensive phones with larger screens. An interesting property of the Skyline operator is that any point p_M that maximizes a monotone scoring function applied on the data, is included in the skyline. Therefore, the example's user will find the best phone according to their preference, regardless of whether this preference is mostly towards cheap phones or phones with big screens. Additionally, every point of the skyline is a maximal point of a monotone function. This means that each of the phones returned from the example's skyline query could match at least one user's specific preference.

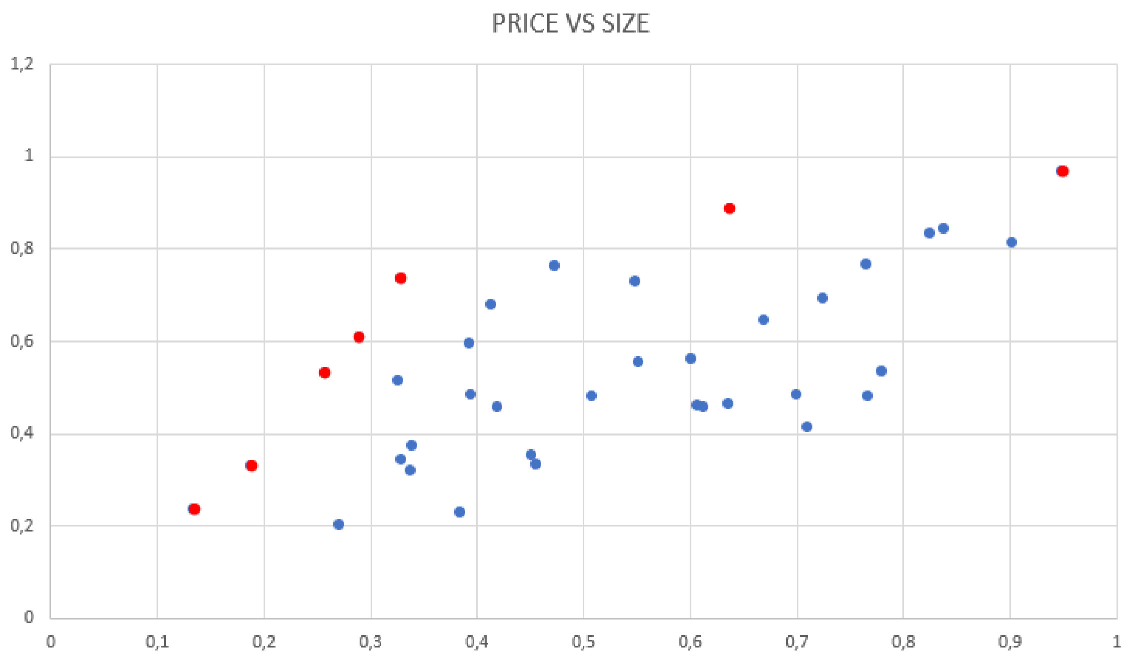


Figure 1 The price versus size skyline points

Although the term skyline calculation was not present earlier, the problem roots to the older mathematical problem of finding the maximal vectors on a set of n d-dimensional vectors in the Cartesian product $U_1 \times U_2 \times \dots \times U_d$.

1.2 The maximal vector computation and the partially ordered set

Maximal vectors have grown a lot of attention to mathematicians between 1970 – 1980 because they compose a set of interesting vectors in a partially ordered set [2][3]. Partial orders help generalizing the concept of total orders, (where a binary comparison exists between every element, e.g. one-dimensional sets), to multidimensional sets. In a partially ordered set, there exists a comparison between two elements, $x \geq y$, if and only if $x_i \geq$

y_i for every dimension i. Thus, not all elements of the set are comparable. The relation $\leq$ in a partially ordered state is reflexive, antisymmetric, and transitive. [30]

An element x is a maximal vector of a set when exists no y for which $y \geq x$. For example, between the $(2,3,1), (4,2,3), (1,6,3), (2,4,3)$ elements of a set, (2,3,1) is the only element which is not a maximal vector. The computational complexity of the maximal vector problem is calculated to be

- $C_d(n) \leq O(nlog_2 n)$ for $d = 2, 3$,
- $C_d(n) \leq O(n(\log_2 n)^{d-2})$ for $d \geq 4$, and
- $C_d(n) \geq O(\lceil \log_2 n! \rceil)$ for $d \geq 2$. [1]

The algorithms for calculating the maximal vector can be easily adjusted to return the minimal vector of a set. The union of those two sets produces the convex hull of the set which is relevant to several problems in areas like computer graphics, design automation and pattern recognition.

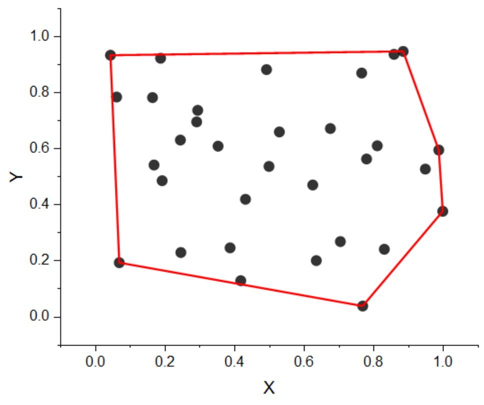


Figure 2 Example of a 2d convex hull graph (source: https://www.originlab.com/fileExchange/details.aspx?fid=355)

One-dimensional sets can be easily calculated after $n - 1$ comparisons (n being the number of elements) using a max operator. 2-dimensional sets can be also computed effortlessly by firstly pre-sorting the set according to one dimension. Thus, the skyline computation mostly concerns > 2 −dimensional datasets.

1.3 Skyline computation in the Apache Spark platform

Designing an efficient skyline calculation algorithm becomes gradually more difficult as the volume of data increases. For a limited size of data, a simple SQL query is able to return results in a satisfying period of time. When the input of the algorithm becomes bigger, more sophisticated algorithms must be introduced. Moreover, in the era of Big Data, a single processing machine cannot always handle such calculations and the need for algorithms that are designed for distributed execution are more than necessary.

This thesis aims to implement efficient skyline calculation algorithms in Apache Spark, a cluster-based platform for parallel and distributed programming. Those algorithms are designed with respect to the skyline literature but are adjusted to the unique architecture of Apache Spark. In chapter 2 previous research conducted on the Skyline problem will be introduced. The literature's algorithms, designed for single machines, distributed environments and Apache Spark, will be analyzed while the characteristics of a distributed computing system are set forth. Chapter 3 pertains to the Apache Spark platform, focusing on Spark's architecture and programming environment. Inside chapter 4, the algorithms designed for this thesis are analyzed. Spark's execution plans formed for those algorithms are shown and optimization techniques used to improve the algorithms' efficiency are described. In chapter 5 the algorithms' performance in a single unit and in the Hadoop environment is recorded, discussing each algorithms' results. The final chapter contains the thesis' conclusions regarding the Spark architecture's effect on the process of designing an efficient Skyline calculation algorithm as well as suggestions for future work.

2 Literature review on the skyline calculation

Stephan Borzsonyi et al. [1] were the first to propose several algorithms for constructing a skyline operator as an extension of the core SQL operators. Those algorithms were able to consider only specified attributes of the database and their domination preference (min or max). The user of our example would be able to ignore other attributes like weight and battery consumption and aim to minimize the price and maximize the screen size.

Borszonyi presented a baseline Skyline nested SQL query, composed by the core SQL operators, stating that it performs poorly on a large amount of data, and new algorithms need to be developed for the skyline problem. The algorithms proposed are based on block-nested loops and divide-and-conquer methods, while the use of R-trees is also shortly introduced.

The block-nested algorithm uses the driver's memory to temporarily store non-dominated points that are then compared and replaced (in case of domination) from an incoming, previously unexamined point. Timestamps are used to determine the order of the comparisons and temporary files to store candidate skyline points in case of memory overloads.

The Divide-and-Conquer algorithm recursively partitions the dataset based on the median of some dimension until a partition contains one or a few points and the skyline computation is easily applied. The skyline of the whole dataset is obtained by recursively merging those partitions while eliminating dominated points

```
SELECT * FROM Hotels h
WHERE h.city = 'Nassau' AND NOT EXISTS(
SELECT * FROM Hotels h1 WHERE h1.city = 'Nassau'
AND h1.distance <= h.distance AND h1.price <= h.price
AND (h1.distance < h.distance OR h1.price < h.price));
```

Script 1: example of skyline computation using nested SQL [1]

M	Input; a set of d-dimensional points
R	Output; a set of d-dimensional points
T	Temporary file; a set of d −dimensional points
S	Main memory; a set pf d −dimensional points
$p \prec q$	Point p is dominated by point q

```
function SkylineBNL($M$)
begin
//initialization
$R \coloneqq \emptyset, T \coloneqq \emptyset, S \coloneqq \emptyset$
$CountIn \coloneqq 0, CountOut \coloneqq 0$
//Scanning the database repeatedly
while $\neg EOF(M)$ do begin
    foreach $p \in S$ do
        if $TimeStamp(p) = CountIn$ then $save(R,p), release(p)$
    $load(M,p), TimeStamp(p) \coloneqq CountOut$
    $CountIn \coloneqq CountIn + 1$
    foreach $q \in S \backslash \{p\}$ do begin
        if $p \succ q$ then $release(p)$, break
        if $p \prec q$ then $release(q)$
    end
    if $\neg MemoryAvailable$ then begin
        $save(T,p), release(p)$
        $CountOut \coloneqq CountOut + 1$
    end

    if $EOF(M)$then begin
        $M \coloneqq T, T \coloneqq \emptyset$
    end
end
//Flushing the memory
foreach $p \in S$ do $save(R,p), release(p)$
return $R$
end
```

Script 2: the BNL algorithm [1]

```
function SkylineBasic($M, Dimension$)
begin
if $|M| = 1$ then return $M$
$Pivot \coloneqq Median\{m_{Dimension} | m \in M\}$
$(P_1, P_2) \coloneqq Partition(M, Dimension, Pivot)$
$S_1 \coloneqq SkylineBasic(P_1, Dimension)$
$S_2 \coloneqq SkylineBasic(P_2, Dimension)$
return $S_1 \uplus MergeBasic(S_1, S_2, Dimension)$
end

function MergeBasic($S_1, S_2, Dimension$)
begin
if $S == \{p\}$ then $R \coloneqq \{q \in S_2 | p \nprec q\}$
else if $S_2 = \{q\}$ then begin
    $R \coloneqq S_2$
    foreach $p \in S$ do if $p \prec q$ then $R \coloneqq \emptyset$
end else begin
    $Pivot \coloneqq Median\{p_{Dimension-1} | p \in S_1\}$
    $S_{1,1}, S_{1,2} \coloneqq Partition(S_1, Dimension - 1, Pivot)$
    $S_{2,1}, S_{2,2} \coloneqq Partition(S_2, Dimension - 1, Pivot)$
    $R_1 \coloneqq MergeBasic(S_{1,1}, S_{2,1}, Dimension)$
    $R_2 \coloneqq MergeBasic(S_{1,2}, S_{2,2}, Dimension)$
    $R_3 \coloneqq MergeBasic(S_{1,1}, S_2, Dimension - 1)$
    $R \coloneqq R_1 \uplus R_3$

end
return $R$
end
```

Script 3: Divide-and-Conquer algorithm [1]

During the same year Kian-Lee Tan et al. [4] proposed *Bitmap* and *B+-tree* based algorithms that produced, in contrast to BNL and DC, progressive results. The first, converts each tuple p to a sequence of m bits. Those bits are calculated based on the total distinct values each dimension contains throughout the dataset. The bitmaps are then stored and processed as bit-slices. The decision over a skyline point is calculated much more efficiently because the calculations are conducted on bits. Moreover, a point calculated to be a skyline point can be instantly output as such and deleted from memory and further consideration.

The B+-tree based algorithm transforms and maps multi-dimensional into one-dimensional data. B+-trees are used to index the transformations. That results to excluding points that are obviously dominated as well as producing some skyline points in a short period of time.

```
foreach point $x = (x_1, x_2, \dots, x_d)$ in the database
let $x_i$ be the $q_i$th distinct value in dimension $i$
$A \coloneqq BitSlice(q_1, 1)$
for $i = 2$ to $d$
    $A \coloneqq A \,\&\, BitSlice(q_1, i)$
$B \coloneqq BitSlice(q_1 - 1, 1)$
for $i = 2$ to $d$
    $B \coloneqq B \mid BitSlice(q_1 - 1, i)$
$C \coloneqq A \,\&\, B$
If $C == 0$
    Output $x$
```

Script 4: Bitmap algorithm [4]

```
for $i = 1$ to $d$
    $f_i \coloneqq True$
    $t_i \coloneqq traverseTreeMax(root, i)$
    $max_i \coloneqq maxValue(t_i)$
    $min_i \coloneqq minValue(t_i)$
$mn \coloneqq max_{i=1}^{d} min_i$
$mx \coloneqq max_{i=1}^{d} max_i$
for $i = 1$ to $d$
    if $mn > max_i$
        $f_i \coloneqq False$
$j \coloneqq 1$
$S \coloneqq \emptyset$
while there are some partitions to be searched
    for $i = 1$ to $d$
        if $max_i == mx$
            $P_j \coloneqq t_i$
            $S_j \coloneqq \emptyset$
            $t_i \coloneqq getNextLeftElement(t_i)$
            while $(maxValue(t_i) == mx)$
                $mn \coloneqq \max(mn, minValue(t_i))$
                $P_j \coloneqq P_j \cup t_i$
                $t_i \coloneqq getNextLeftElement(t_i)$
            $max_i \coloneqq maxValue(t_i)$
        $S_j \coloneqq computePartitionSkyline(P_j)$
        $S \coloneqq S \cup computeNewSkyline(S_j, S)$
        $j \coloneqq j + 1$
        $mx \coloneqq max_{i=1}^{d} max_i$
        for $i = 1$ to $d$
            if $mn > max_i$
                $f_i \coloneqq False$
```

Script 5: B+-tree-based algorithm [4]

At 2002 Donald Kossmann also focused on progressive skyline computation, and more specifically on online implementations [5]. In these cases, the user focuses on receiving the first skyline points in an efficient period and does not demand the whole skyline vector until they investigate those points. In contrast to the Bitmap and B+-tree based algorithms, Kossmann designs an algorithm that returns fair early results. That is, results that are balanced and not in favor of one specific dimension. In addition, it provides the possibility to the user to adjust their preferences while the algorithm is running (to accelerate the return of skyline points that are neighbors of a returned one). It uses Nearest Neighbor methods to partition the dataset into regions and to exclude those regions that are evidently dominated by other.

```
Input: Dataset $D$
       Distance function $f$ (e.g., Euclidean distance)
$T \coloneqq \{(-\infty, \infty)\}$
while $T \neq \emptyset$ do
    $(m_x, m_y) \coloneqq takeElement(T)$
    if $\exists\, boundedNNSearch(O, D, (m_x, m_y), f)$ then
        $(n_x, n_y) \coloneqq boundedNNSearch(O, D, (m_x, m_y), f)$
        $T \coloneqq T \cup \{(n_x, m_y), (m_x, n_y)\}$
        return n
    end if
end while
```

Script 6: NN-based algorithm [5]

Some papers proposed alterations on the previous algorithms to further improve optimization. Dimitris Papadias, in his paper “An Optimal and Progressive Algorithm for Skyline Queries” [6] also uses a tree-based Nearest Neighbor technique that avoids redundant calculations. Jan Chomicki [7] proposed pre-sorting the tuples before performing BNL, taking into advantage the fact that in a pre-sorted dataset, a tuple cannot be dominated by subsequent tuples. Other papers focus on different types on databases. [8] focuses on streaming data. [9],[10] and [11] propose algorithms for uncertain data (due to measurement/quantization errors, data staleness, and multiple repeated measurements etc.).

All the above approaches offer effective results when applied on traditional RDBM systems, where data are stored and processed from single machines. The architecture of those machines though creates limitations regarding the volume of data they can store and process in main memory as well as the coordination of the processes. Figure 3 shows that since 2004, skyline computation in distributed environments is an emerging field of research and will be discussed further in the next paragraph.

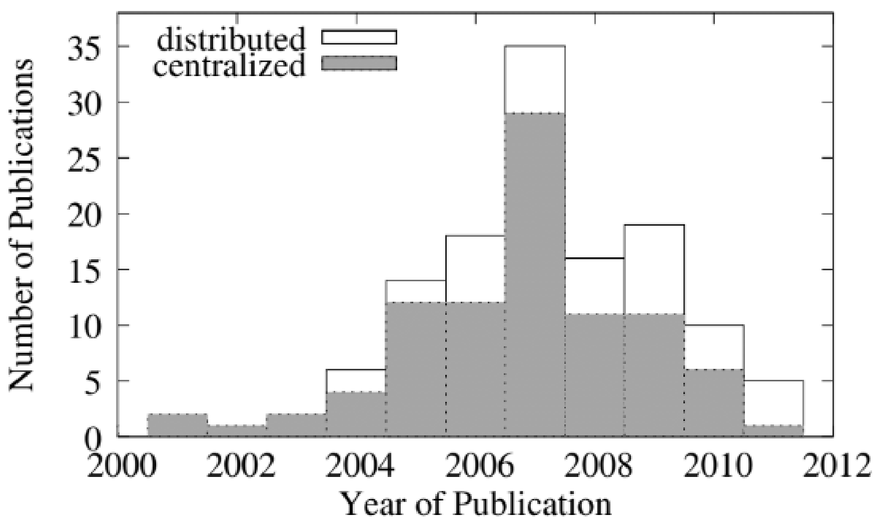


Figure 3: distributed and centralized Skyline publications per year [12]

2.1 Skyline computation in distributed environments

Before reviewing the recent research on skyline algorithms for distributed systems, it is important to clarify what a distributed system is.

2.1.1 Distributed computing systems

While many definitions exist until now, their common ground is that distributed systems require the use of multiple processors. This paper will follow the definition of Henri E. Bal et la [13]:

> *Definition. "A distributed computing system consists of multiple autonomous processors that do not share primary memory but cooperate by sending messages over a communications network."*

The subject of the distribution varies among different architectures. Some systems distribute processing logic and elements, while others distribute tasks based on the function of the system's hardware unit (printers, fax, etc.) Based on this definition the processors of a distributed system do not share primary memory. This differentiates the systems' processors from multi-processors (processors that share the same memory) and therefore distinguishes the terms distributed and parallel processing, although in many cases, the distributed processors use parallel computations. The types of communication networks between the distributed systems' processors vary among different architectures. Closely coupled distributed systems contain processors physically near each other, therefore their communication cost is minimal. Loosely coupled systems are set in LAN workstations or even more globally set in WAN networks, like the Internet.

Distributed systems provide many benefits as opposed to local systems. They offer higher performance, due to the parallel execution over multiple processors depending on the volume of the dataset and the task an application is set to execute. They provide higher fault tolerance in the case of a processor's failure. While the risk of this failure is low, it can sometimes be critical and lead to data loss and require the termination and restart of the application. Contemporary distributed systems often provide duplications of data between the processors, and a partial failure of one processor does not affect the functioning of the others, while the lost data can be replaced instantly by their duplicates. Moreover, some applications require exclusively the use of a distributed environment, for example multi-national company applications and email services. [13]

2.1.2 Distributed file systems

Distributed computing systems initiated the need for sharing data mechanisms across the multiple processors. The first limited and inconvenient approach was to use user-initiated file transfer for remote file access. Until the early '80s the distributed file systems literature started to recognize the need of resembling the local filesystem user experience (network transparency). A major evolution breakthrough came with LOCUS, a discontinued distributed filesystem which was created at UCLA between 1980 – 1983. The two innovative properties of LOCUS were the location transparency and the data replication, as a fault tolerance method. [14]

The Hadoop Distributed File System

Today, one of the most preferred DFSs for distributed computing is the *Hadoop Distributed File System (HDFS)*. Together with the *Hadoop MapReduce*, the Hadoop Ecosystem was created as an open source alternative of Google's File System and MapReduce, that were used as a model for processing and generating large data sets.

HDFS follows a master/slave architecture, consisting of a Namenode, several Datanodes, and the HDFS client. The NameNode is responsible of the namespace structure and the filesystem metadata. The Datanodes store the HDFS data in the form of blocks in the local file systems. The blocks that are physically close are organized in racks. The Datanodes receive commands from the Namenode for data block replication, removal of replicas, re-registrations or shutdowns and reporting node information to the Namenode. The HDFS client is responsible for exporting the HDFS file system interface to applications, reading data directly from Datanodes and setting node-to-node pipelines in which it enters data and writes the output result. HDFS uses block replication as a fault-tolerance method assuring by default that none of the Datanodes contains more than one replica of any block and none of the racks contains more than two replicas of the same block.

Hadoop MapReduce, like Google's MapReduce, is a software framework for processing and generating large data sets. It most effectively performs on top of the HDFS. A job in MapReduce separates the data into chunks and using map performs parallel tasks in each of them. The outputs of the chunks are sorted by the framework and then proceed to the reduce tasks. The MapReduce framework forces the input and output to be formed as $< key, value >$ pairs sets. [15]

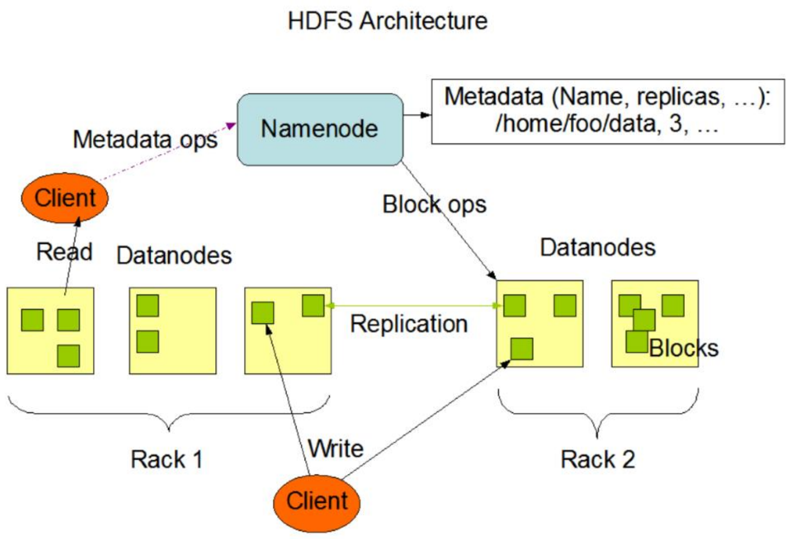


Figure 4: HDFS architecture [15]

2.1.3 Distributed skyline algorithms

Skyline computation in a distributed environment offers important advantages. The size of the dataset is not restrictive because it does not need to fit into a single machine's memory. It is easily scalable; the integration of an additional node into the system requires less cost and effort than to upgrade the core machine's hardware. Most importantly, parallel processing decreases significantly the computational cost of an algorithm. Local-based algorithms though, do not exploit the above benefits, being optimized for the hardware characteristics of a single machine. The research of skyline algorithms that apply well in distributed systems has gained a lot of attention, aiming to decrease the high processing cost of a skyline algorithm.

According to A. Vlachou [12], all the distributed skyline algorithms literature aims to minimize the execution time of the algorithm taking into consideration the total processing time, the number of queried peers and the network traffic of the execution and the contradiction towards each other.

A baseline approach proposes the horizontal partitioning of the dataset into chunks and locally calculating the skyline points of each chunk. The results are then collected by the coordinator which calculates the final skyline tuples. This approach, called all local skylines (ALS) [16] does not guarantee that the local skylines are few enough to fit and be processed in main memory. Additionally, the algorithm calculates and transfers all the local skylines, without using smart methods to distinguish those that are dominated by tuples of another partition. This leads to expensive bandwidth consumption. The effectiveness of this approach depends on the local and centralized skyline algorithms used.

In 2006, Zhiyong Huang et al. was the first to research skyline algorithms in non-centralized, share-nothing systems [17]. His paper concerned constrained skyline querying in distributed mobile systems and more specifically in wireless mobile ad hoc networks (MANETs). By using a breadth-first approach, the device that produces the query, sends it to all its neighbors which then return their local skylines and transfer the query to their neighbors. If a depth-first approach is used, the querying device sends the query to only one of its neighbors which propagates it to one of their neighbors. The skyline tuples are collected once the device does not find another neighbor and are being merged through the same path. For further traffic optimization, along with the query, a significant tuple is sent to the next device to help pruning their local dataset.

```
algorithm local_skyline($pos_{org}, tp_{flt}$)
input: $pos_{org}$ is the location of the query originator
       $d$ is the distance of interest
       $tp_{flt}$ is the filtering tuple
Output: reduced local skyline and updated filtering tuple

If $(mindist(pos_{org}, MBR_i) > d)$ return;
$skip := True$
foreach attribute $j$ of $R_i$
    If $(tp_{flt}.p_j > l_j)$ $skip := False$ break;
if $(skip)$ return; else $SK_i := \emptyset$
foreach tuple $tp_j$ in $R_i$
    if $(dist(pos_{org}, tp_j) > d)$ continue;
    $out := False$
    foreach skyline point $sp_k$ in $SK_i$
        if $\forall l > 1, sp_k.id_l < tp_j.id_l)$ $out := True$ break;
    if $(!out)$ add $tp_j$ into $SK_i$
$idx := null, VDR_M := 0$
foreach skyline point $sp_k$ in $SK_i$
    if $(\forall l, tp_{flt}.p_t < sp_k.pl)$ remove $sp_k$ from $SK_i$
    else if $(VDR_k > VDR_m)$ $idx := k,$ $VDR_m = VDR_k$
if $(VDR_m > VDR_{flt})$ $tp_{flt} = tp_{idz}$
```

Script 7: Zhiyong Huang’s algorithm [17]

Vlachou et al. introduced *Skypeer* [18], a subspace skyline query algorithm for peer-to-peer systems that consist of many peers and fewer super-peers (peers with enhanced capabilities). The algorithm aims to decrease the workload of the simple peers by relying on the super-peers. The term ext-domination and ext-skyline are presented. Ext-domination of a tuple q by a tuple p exists when $p[i] >$ (instead of $\geq$) $q[i]$ for each dimension i of the set. It also uses mapping functions and thresholds to further optimize the algorithm.

```
Algorithm 1 local subspace skyline computation
input: $U$ is the location of the query originator
$SKY_U \coloneqq \{\emptyset\}$
$threshold \coloneqq MAX\_INT$
$p \coloneqq next\ point\ based\ on\ f(p)$
while $(f(p) < threshold)$ do
    if $p$ is not dominated by any point in $SKY_U$ based on $U$ then
        remove from $SKY_U$ the points dominated by $p$
        $SKY_U \coloneqq SKY_U \cup \{p\}$
        $threshold \coloneqq min_{p_i \in SKY_U}(dist_U(p_i))$
    end if
    $p \coloneqq next\ point$
end while
return $SKY_U$
```

```
Algorithm 2 Super-peer merging of subspace skylines
input: $U$ denotes the query dimensions
$SKY_U \coloneqq \{\emptyset\}$
$threshold \coloneqq MAX\_INT$
$SKU_{U_1} \dots SKY_{U_{N_{sp}}}$ the super-peers' set of local subspace skyline points
$SKY_{U_a} \coloneqq$ the list whith the minimum first element
$p \coloneqq$ next point based on $SKY$
$p \coloneqq next\ point\ based\ on\ f(p)$
while $(f(p) < threshold)$ do
    if $p$ is not dominated by any point in $SKY_U$ based on $U$ then
        remove from $SKY_U$ the points dominated by $p$
        $SKY_U \coloneqq SKY_U \cup \{p\}$
        $threshold \coloneqq min_{p_i \in SKY_U}(dist_U(p_i))$
    end if
    $p \coloneqq next\ point\ based\ on\ SKY$
end while
return $SKY$
```

Script 8: The Skypeer algorithms [18]

The paper of Bin Cui et al. presents PaDSkyline [19], which aims to optimize distributed constrained skyline queries in a network environment without assuming any overlay structures. PaDSkyline at first uses MBR (n-dimensional minimum bounding box of the local relation Ri) to partition the dataset into incomparable groups and eliminate groups that disjoint with the query's constrains.

DSL (Distributed SkyLine) proposed by Wu et al. [20] partitions the tuples based on regions using a multi-level hierarchy. The low-level partitions calculate the local skylines which are then merged to the higher-level partition. The data partitioning is determined by CAN, a distributed, decentralized P2P infrastructure, based on a logical d-dimensional Cartesian coordinate space, which incorporates a distributed hash table (DHT) for point and server multi-dimensional indexing. Next an intra-group query execution takes place in each group.

```
Algorithm icmpPartition($S, C$)
input: $S$ is the set of data sites
       $C$ is the set of constrains in the skyline query
Output: an incomparable partition of $S$
foreach $S_i \in S$
    $rMBR_i := MBR_i \cap C_i$
    If ($rMBR_i == \emptyset$)  $S := S - \{S_i\}$
$\prod S = \{\{S_{1'}\}\}$ // $S_{1'}$ is the current 1st element in $S$
foreach $S_i \in S - \{S_{1'}\}$
    $\overline{S_\iota} = \emptyset$
    foreach $S_i \in \prod S$
        if ($\exists S_j \in S_i$ $s.t. S_j$ $and$ $S_i$ $are$ $not$ $incomparable$)
            $\prod S = \prod S - \{S_i\}$;  $\overline{S_\iota} = \overline{S_\iota} \cup S_i$
        $\prod S = \prod S \cup \{\{S_i\} \cup \overline{S_\iota}\}$
```

Script 9.1: group partitioning phase of PaDSkyline [19]

```
Algorithm groupSkyline($C, S_{org}, plan$)
input: $S_{org}$ is the query originator site identifier
       $C$ is the set of constrains in the skyline query
       $plan$ is the query execution plan in the group
Output: the constrained skyline within the group
Compute local skyline $R_g$ and get the initial filtering points set $F_c$
Send $\langle C, S_g, plan', F_c \rangle$ to next site(s) in $plan$
repeat
    Receive result reply from a group member $S_i$
    Merge $S_i.R_i$ with $R_g$, remove duplicates and false positives
until all group members have replied
return $R_g$ to $S_{org}$
```

Script 9.2: local skyline execution of PaDSkyline [19]

```
Algorithm PaDSkyline($S, C$)
input: $S$ is the set of data sites
       $C$ is the set of constrains in the skyline query
Output: the constrained skyline
$\prod S := icmpPartition(S, C)$
foreach group $g_i \in \prod S$ in parallel
    send $\langle C, S_{org}, g_i, plan \rangle$ to $g_i$'s group head
repeat
    receive result reply from a group $g_i$'s head
    report $g_i.result$
until all group heads have replied
```

Script 9.3: The PaDSkyline algorithm [19]

Wang et al.'s objective was the proposition of effective distributed skyline queries in BATON networks. The peers organized in a binary tree and each peer is responsible for a certain region of the dataspace. Load balancing is achieved by splitting and merging techniques and sampling. Later the algorithm was generalized further with *Skyframe* [22].

2.1.4 Distributed Skyline computation in Apache Spark

Although Spark is one of the most popular frameworks for parallel data processing, few attempts have been made in literature for implementing a skyline operator in Spark. Spark's architecture differentiates from those of the systems used in the publications mentioned in the previous chapter. One of the core differences is that Spark's nodes do not exchange information with each other, but all the communication appears between the cluster's manager and the nodes. In addition, Spark implements optimization techniques during the execution of the scripts.

In 2015, a skyline operator was introduced for Spark as part of a correlation framework for spatio-temporal events [24]. It followed the ideas presented in [23]. A grid partitioning schema is created which is then represented as a *bitstring*. This is then used to prune the data of the partitions. The details of this implementation are not described in literature. In 2016 Konstantinos Paparidis [25] evaluated the processing time of angle, random and grid partitioning skyline algorithms in Apache Spark using the standalone deploy mode.

3 Spark

Apache Spark started as a research project at UC Berkeley in the AMPLab, which focuses on big data analytics. Spark's goal is to expand the *MapReduce* capabilities while still being a highly fault-tolerant cluster computing framework. The main disadvantage of MapReduce is that it uses acyclic data flow. The distinctive jobs are run sequentially and between them, the jobs' input/output is read from and written to the stable memory, increasing the I/O cost. In Spark, on the other hand, the data are transferred in-memory between transformations, which makes it efficient for data mining, iterative programming and streaming applications. [31]

3.1 Spark architecture

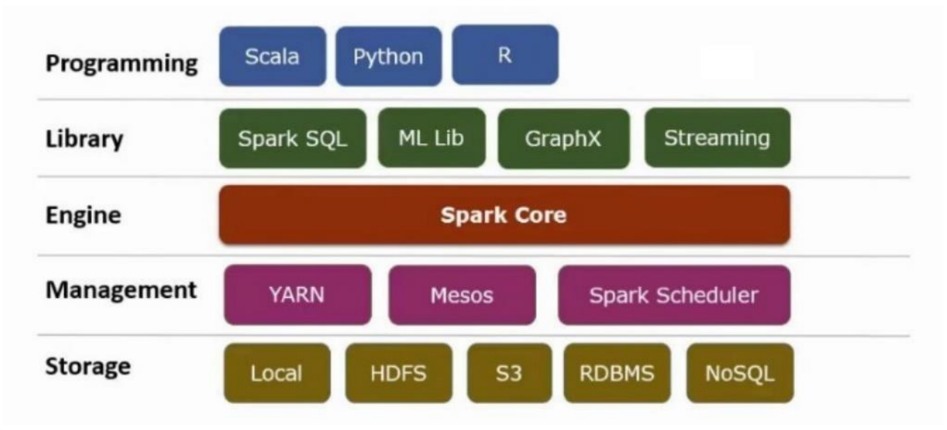


Figure 5 The Spark Architecture (source: https://www.youtube.com/watch?v=ZTFGwQaXJm8)

3.1.1 Programming Languages

Spark's early versions (2012) were written exclusively in *Scala*, a concise and fast programming language that is both object-oriented and functional. Scala is statically typed and interoperates well with the Java Runtime Environment. In 2013 a *Python API* was included in the Spark Core release and since 2015 Spark provides an API for the *R programming* language.

3.1.2 Spark Libraries

Spark provides four libraries, each serving different purposes:

Spark SQL, first released in 2014, provides a DataFrame API for relational operators that can accept SQL queries, offering high optimization level due to Spark's lazy evaluation. It introduces Catalyst, an extensible optimizer through which a variety of data sources can be used including semi-structured JSON data, and data manipulation is possible via user-defined functions. (source: https://spark.apache.org/sql/)

GraphX is used for graphs and graph-parallel computation. It offers an abstraction extending Spark's RDD abstraction which is discussed in the next paragraphs, named Resilient Distributed Graph which links records with vertices and edges in a graph and provides a set of graph computations. (source: https://spark.apache.org/graphx/)

MLlib is a library for distributed machine learning. It consists of a variety of broadly used machine learning algorithms written in a scalable and fast manner, taking into advantage the parallelisms of Spark. (source: https://spark.apache.org/mllib/)

Spark Streaming provides scalable and fault-tolerant processing of data streams. It divides the input streams into batches that are then processed using Spark's functionalities, and lead to the result streams. This library contains a high-level abstraction called *discretized stream* (DStream) that is internally represented as a collection of RDDs. (source: https://spark.apache.org/streaming/)

3.1.3 Spark Execution

Spark applications are coordinated by the SparkContext object in the user's driver (main) program. SparkContext is then connected to the cluster manager, which is responsible for the resource allocation. Via the manager, Spark sets executors that run computations and store local data on each node. Finally, it sends the driver's code to the executors and SparkContext sends tasks to the executors to run. (source: https://spark.apache.org/docs/latest/cluster-overview)

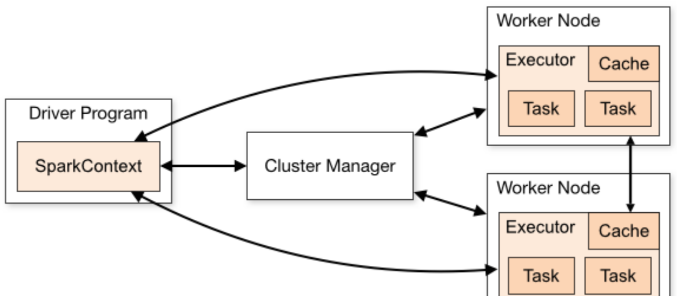


Figure 6 Spark execution (source: https://spark.apache.org/docs/latest/cluster-overview.html)

3.1.4 Cluster Managers

The Spark engine is unable to identify the cluster manager responsible for the application. Currently four different cluster managers are supported. The *standalone manager* is integrated in Spark and it allows the application to be deployed in cluster mode if Spark is built in each of the cluster's nodes. Moreover, the application can be deployed locally, in a single machine, for testing and debugging reasons. The *Apache Mesos* cluster manager, which supports Hadoop MapReduce, enables building and running applications in a distributed system by abstracting CPU, memory and other resources from machines. Mesos can perform dynamic resource allocation between Spark and other frameworks as well as dynamically scale the application's partitions. *Hadoop Yarn manager*, which supports Hadoop 2, separates the functionalities of resource management and job scheduling/monitoring into different daemons. Spark can also be built on *Kubernetes*, an open-source system for automating deployment, scaling, and management of containerized applications, although this manager is still in experimental mode. (source: https://spark.apache.org/docs/latest/cluster-overview)

3.1.5 Storage Systems

Like cluster managers, Spark is agnostic regarding the storage system used in an application. This allows the processing of existing data as well as the combination of data from different data sources. It can use local, distributed file systems (like HDFS), key-value stores like S3 and Cassandra and also connect with Apache Hive as a data catalogue.

3.2 Spark programming environment

Spark's two main abstractions are resilient distributed datasets (RDD) and parallel transformations applied on them.

3.2.1 RDDs

RDDs are fault-tolerant, parallel data structures that represent a read-only collection of objects partitioned across a set of machines. The intermediate results of RDD calculations are stored and processed in-memory, which leads to massive improvement of the applications' performance. A user is equipped with a large amount of available RDD operations and can control an RDD's persistence and partitioning procedure. They are based on coarse-grained transformations like map, reduce, filter etc. Spark uses lazy evaluation for these transformations, seeking an efficient plan for implementing the user's instructions; The transformations return an RDD object which represents the transformation's result without processing the data. Only when an action is set on an RDD, Spark creates an execution plan for the transformations that result to it. Actions are operators that return a result in the base memory or write into the storage system. In a case of a node failure, Spark uses the transformation pipeline log rather than the actual data, to revive the data in a former safe state. This improves the fault tolerance of the application. Internally, each RDD is characterized by five main properties:

- A list of partitions
- A function for computing each split
- A list of dependencies on other RDDs
- Optionally, a Partitioner for key-value RDDs
- Optionally, a list of preferred locations to compute each split on

(source: https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/rdd/RDD.scala)

A word count example

```
val text = sc.textFile("mytextfile.txt")
val counts = text.flatMap(line => line.split(" "))
.map(word => (word,1)).reduceByKey(_+_).collect()
```

Script 10: Example of a word count using Spark RDDs in Scala

- In the first line, a local file is parallelized into an RDD object according to the Spark configuration. Each element of this RDD is a line of the text.
- Next, each line of the RDD is split into different words and the RDD now consists of 'word' elements.
- In the last line, each word is transformed into a $< key, value >$ pair where key is the word and value is the integer '1'. All the pairs that share the same word (key) are then aggregated, having as value the addition of their separate values.
- Finally, all the RDD elements (different text words) are returned to the driver.

Creating an RDD object

RDD variables can be created by pointing to an existing RDD of the cluster, distributing a main-memory Array-like object or distributing a file that is placed in the driver.

```
val data = Array(1, 2, 3, 4, 5)
val distData = sc.parallelize(data) // RDD from Array object
val distFile = sc.textFile("data.txt") // RDD from textFile
val newRDD = distFile.map( x => x + 1 ) // RDD from another RDD
```

Script 11: different ways of creating an RDD

In the first two examples, the SparkContext object separates the data into blocks and creates a partition for each one of them. By default, each partition is 128MB. If the programmer decides to split the data into more partitions, they add the number of partitions as an argument to the function.

RDD transformations

Transformations are functions that are acted upon an RDD object and return one or more RDD objects. RDDs are immutable objects, which means that the transformations do not change the parent RDD object they are acted upon but result to another child RDD(s) containing the applied changes. By applying multiple transformations, a *Directed Acyclic Graph (DAG)* of transformations is built for all the RDDs that resulted to the final RDD. The DAG is used as a logical execution plan.

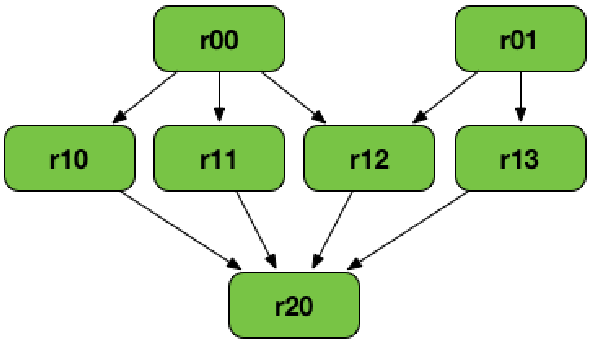


Figure 7 linage of RDD objects (source: https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-rdd-lineage.html)

One way to categorize transformations is to divide them into *narrow* and *wide* transformations. Narrow transformations only require processing the data of a single partition. Spark interprets a sequence of narrow as a pipeline that results to a single stage to be executed. Examples of narrow operators are map and filter. Wide transformations may need data from more partitions to be executed. Spark must execute a *shuffle* event to reform the partitions of the RDD. Shuffle is used when regrouping is necessary across the partitions. This operator is costly and complex since it containts disk I/O, data serialization, and network I/O and should be used only if necessary. Examples of wide transformations are *groupByKey* and *reduceByKey*.

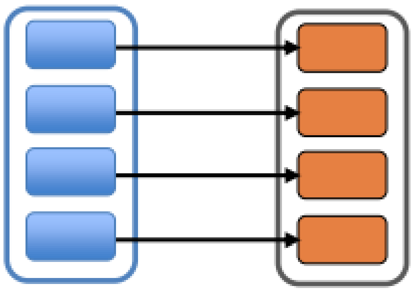


Figure 8 Example of a narrow transformation blue: partition of parent RDD, orange: partition of child RDD (source: Transformations and actions a visual guide training http://training.databricks.com/visualapi.pdf)

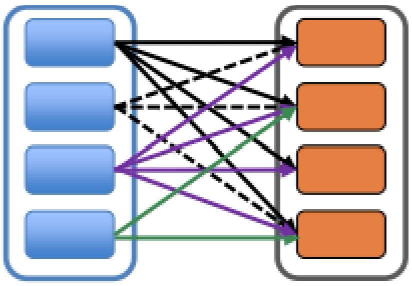


Figure 9 Example of a wide transformation (source: Transformations and actions a visual guide training http://training.databricks.com/visualapi.pdf)

Most common transformations:

General

- map(*func*): a function is executed for each element of an RDD object and the result is another distributed RDD object
- filter(*func*): results to an RDD object that does not contain those elements that when inputted in the function return False.

- flatmap(*func*): It has the same logic as *map* but returns the result of a function as a single element. For that reason, the function should return an array type rather than a single element.
- mapPartitions(*func*): performs an action to the elements of each partition. The output is the transformed partition.
- reduceByKey(*func*, *[numPartitions]*): The input is a set of (K, V) pairs and the function aggregates the V values that share the same key (K). The type of the function is $(V, V) \rightarrow V$. Optionally, the result is repartitioned to the number of partitions.

Math / Statistical

- sample(*withReplacement*, *fraction*, *seed*): Operates like any sample operator, returning a random fraction of the elements with, or without replacement.

Set Theory / Relational

- union(*otherDataset*): Returns the pairs (A, B) of two datasets, A and B
- intersection(*otherDataset*): returns the intersection of two datasets A and B

Data Structure / I/O

- coalesce(*numPartitions*): reduces the number of partition of an RDD object.
- repartition(*numPartitions*): Performs shuffling of the elements and repartitions them to a given number of partitions. Source: [27]

RDD Actions

Actions are functions that input an RDD and result to non-RDD objects. They trigger the execution of the DAG formed for this RDD object. Since they return a non-RDD object no further transformations can be performed on an action result.

Most common actions

General

- reduce(*func*): Aggregates the elements of an object using a $(A, B) \rightarrow C$ function.
- collect(): Returns to the driver an array of the RDD's elements.
- forEach (*func*): Forces a function to each element of the object.

Math / Statistical

- count(): Counts and returns the number of the RDD object's elements

Data Structure / I/O

- saveAsTextFile(*path*): Writes the object to a local, or HDFS text file depending on the type of the String path. Source: [27]

RDD functions

By observing the list of actions and transformations available, it is noticeable that many operators (like *map*, *filter*, *flatMap*, *mapPartitions*) require a function argument, that specifies how the operator should work. Those functions can be anonymous syntax functions or static functions of a global singleton object.

```
val data = sc.textFile("spark_test.txt")
val flatmapFile = data.flatMap(lines => lines.split(" "))
```

Script 11.1: Anonymous syntax function

In this example, $flatMap$ takes as an argument an anonymous function: $lines \Rightarrow lines.split(" ")$. The left part of the function (before =>) is equivalent to the parameter of a function. The right part is equivalent to the body of a function. Anonymous functions are able to identify the returned argument when it is needed and writing a 'return' command is not necessary. The same example using an external function:

3.2.2 Application initialization

```
def tokenize (lines: String): Array[String] = {
return lines.split() }
val data = sc.textFile("spark_test.txt")
val flatmapFile = data.flatMap(tokenize)
```

Script 11.2: External function

Regardless of the supported programming languages Spark supports, in order to build Spark applications, the programmer needs to add the Spark distribution dependency on the project, with respect to the compatibility of the distribution and the language versions. Inside the script, the *SparkContext*, an object that tells Spark how to access the cluster, is initialized having a *SparkConf* object as an argument. With SparkConf, the programmer can adjust all the parameters needed for the Spark application.

```
val conf = new SparkConf().setAppName(appName).setMaster(master)
Val sc = new SparkContext(conf)
```

Script 12: Spark initialization

Each Spark application can have only one SparkContext object. The *appName* of this example code is the name that the programmer wants to be displayed on the cluster UI. The master contains the URL of the cluster which is managed either by Spark, Yarn, or Mesos. [27]

3.2.3 RDD Persistence

As aforementioned, once the RDD object is created, the user can transform or apply an action on it. The data of the object are not loaded in memory and the transformations are not executed until an action is applied on the object. Then Spark creates computation tasks that run on each node machine and return only the result of the computation. If more than one actions are performed throughout the program's code for a single RDD object, the tasks are re-executed to create each of the actions result. In some cases, Spark is able to persist the intermediate data of the RDD in order not to be recalculated. To optimize the application's performance though, the programmer can demand that an RDD is persisted in memory and un-persist it when it is no longer necessary by calling *rdd.persist()* and *rdd.unpersist()* respectively. [27]

3.2.4 Shared Variables

A user inserts functions to each transformation they use. These functions can use variables in the scope they are created, which in most cases is the worker node where these variables are copied. For further optimization, Spark provides two shared variable abstractions, broadcast variables and accumulators. The former, ensures that a variable which is wrapped in a broadcast object is sent to the workers only once instead of packaging it with every closure. Accumulators are variables which are read only from the driver, which concentrates values from workers using an associative operation. They are useful for counting and summing elements. [27]

4 Algorithmic techniques for distributed skyline computation on Spark

During this chapter the thesis' algorithms for distributed skyline calculation in Apache Spark are described and the RDD transformations and functions used are presented.

4.1 Problem description

The algorithms aim to input a dataset of multidimensional numerical points and return the minimum skyline points taking into consideration the dominance in every dimension. The algorithms aim to provide an efficient and scalable solution without previous knowledge of the dataset's distribution and size. The code is written in Scala using the Spark API. All the proposed algorithms follow the literature's common patterns for designing distributed skyline computation algorithms. In this manner, three stages are performed:

- Partition the dataset into chunks
- Perform skyline calculation in each chunk
- Gather the local skyline tuples locally and perform skyline computation in the driver

4.2 Algorithmic approaches

4.2.1 All Local Skyline

The first algorithm is the implementation of the baseline approach for distributed skyline calculation called ALS [16]. According to it, the skyline points of each node are calculated in parallel and are then returned to the core machine. Then a skyline calculation is performed for all the returned points and the result is output. This approach offers minimum interference for optimization reasons from the programmer and relies on the optimization tools of Spark.

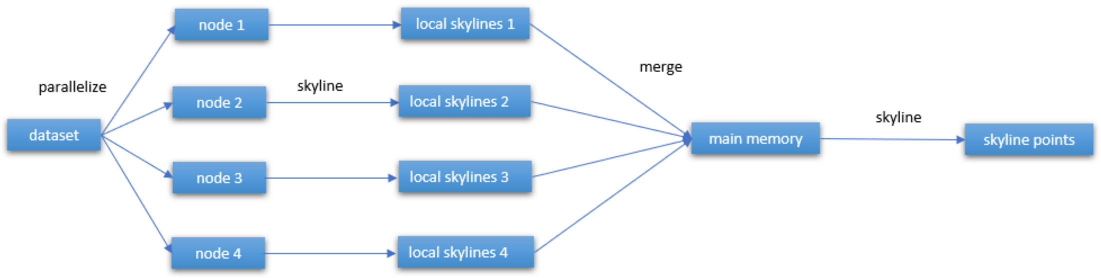


Figure 10 graph representation of ALS in Spark

```
**Algorithm** SparkALS
**Input:** $D$ a multidimensional database
**Output:** $S$ a set of skyline points
$datasetRDD \coloneqq parallelize\ D\ into\ partitions$
**foreach** $partition$ in $datasetRDD$ **do in parallel**
    $partition \coloneqq getSkylinePoints(partition)$
$datasetMain \coloneqq collect\_to\_main\_memory(datasetRDD)$
$S \coloneqq getSkylinePoints(datasetMain)$
```

Script 13: Pseudocode of ALS in Spark

Spark implementation

The first step of the algorithm's implementation in Spark is the parallelization of the input source into an RDD object. Primarily, each element of the source is a $\langle String \rangle$ line containing d float numbers, separated by space (where d is the number of the dataset's dimensions). Through a sequence of map functions this element is split and the $\langle String \rangle$ elements of the split are converted to $\langle Double \rangle$. In order to perform a parallel skyline calculation for each partition as the algorithm dictates, $mapPartitions$ is used and the skyline calculation method is passed as an argument.

```
val rdd2 = rdd
    .map(x=>x.split(" "))
    .map(x => x.map( y => y.toDouble))
    .mapPartitions(skylineCalculation.calculate)
```

The result is an RDD object whose elements are arrays of $\langle Double \rangle$ numbers, that represent the Skyline points of each partition.

Next, those elements are collected in main memory as an $\langle Iterator \rangle$ object and the same skyline calculation method is executed for this object. Finally, the result is written in a *.csv file.

```
val mainMemorySkylines = skylineCalculation.calculate(
    rdd2.collect().toIterator)
 val write = new writeOutputToCSV(mainMemorySkylines, "ALS.csv")
```

There is only one action programmed, therefore, Spark execution manager creates one job having a single stage. The DAG created from the ALS algorithm is displayed below.

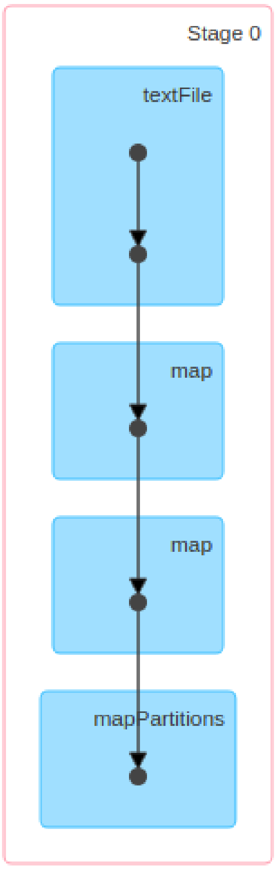


Figure 11: DAG of ALS

4.2.2 Nested SQL Query using Spark SQL

This algorithm explores the functionalities and optimization techniques of Spark SQL. An SQL query for skyline calculation is passed to the Dataframe object, which is then optimized by the Catalyst optimizer that Spark SQL contains. The result of the query is the Skyline Dataframe.

```
Algorithm SparkNestedSQLSkylineCalculation

Input: $D$ a multidimensional database

Output: $S$ a set of skyline points

$datasetRDF \coloneqq parallelize\ D\ into\ a\ Spark\ Dataframe\ Object$

$S \coloneqq datasetRDF.executeSQLQuery($

    "SELECT * FROM dataframe df
    WHERE NOT EXISTS(
    SELECT * FROM dataframe df1 WHERE
    df1.i <= df.i for each dimension i
    AND (df1.i < df.i for at least one dimension i);"
    )

return $S$
```

Spark Implementation

The parallelization of the input source for this algorithm, does not create an RDD object but the Spark SQL's Dataframe abstraction which is handled differently. Initially, after the Dataframe object creation, a database table is formed that includes the object's content. The table is then used for the Skyline SQL query execution. Finally, in order to save the query's result in a single file, the Dataframe object is coalesced into one partition which is then collected in the driver's memory and written in file.

```
df.createOrReplaceTempView("dataset") //table creation

val datasetLength = df.columns.length //extracting the dimensions size

//executing the skyline query

val df2 = confsql.sql(createSkylineQueryString(datasetLength,false))

//writing the result to a space separated csv

df2.coalesce(1)

    .write.format("com.databricks.spark.csv")

    .option("delimiter"," ").save("./output")
```

The skyline query String is formed inside the $createSkylineQueryString$ function. This takes as an argument the number of the dimensions and compares each dimension of the tuples as described in the pseudocode script.

```
def createSkylineQueryString(length: Int, header: Boolean ): String = {
    var query = "SELECT * FROM dataset AS d WHERE NOT EXISTS (" +
      "SELECT * FROM dataset AS d1 WHERE "
    var i = 0
    for (i <- 0 to length -2) {
      query = query + "d1._c" + i + " <= d._c" + i + " AND "
    }
    query = query + "( "
    var j = 0
    for (j <- 0 to length -3) {
      query = query + "d1._c" + j + " < d._c" + j + " OR "
    }
    query = query + "d1._c" + (length-2) + " < d._c" + (length-2) + " ))"
    return query
  }
```

The Catalyst optimizer at first analyses the logical plan of the execution which then seeks to optimize it into a new, more effective plan. The primal logical plan is following the script’s instructions:

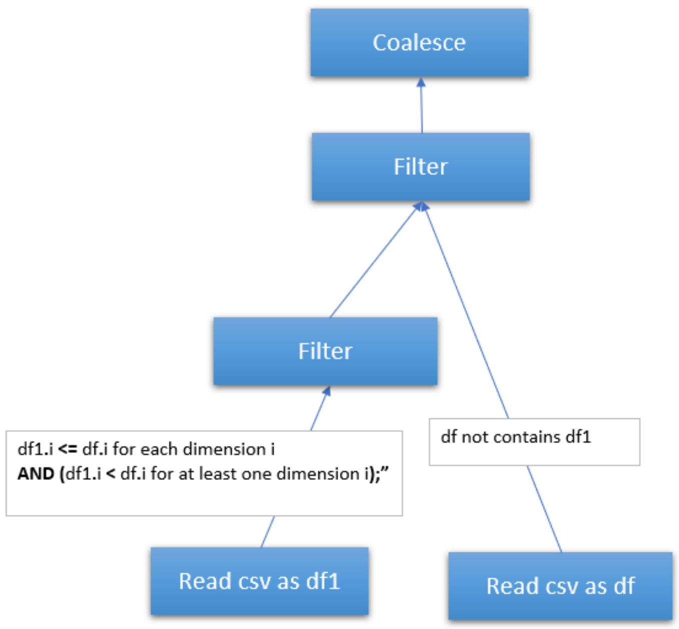

Coalesce
Filter
Filter
df1.i <= df.i for each dimension i
AND (df1.i < df.i for at least one dimension i);”
df not contains df1
Read csv as df1
Read csv as df

Figure 12 DAG of pre-optimization Nested SQL

The optimized plan instead of filters uses *LeftAntiJoin*, a powerful operator that finds values from one table that are not present in another table.

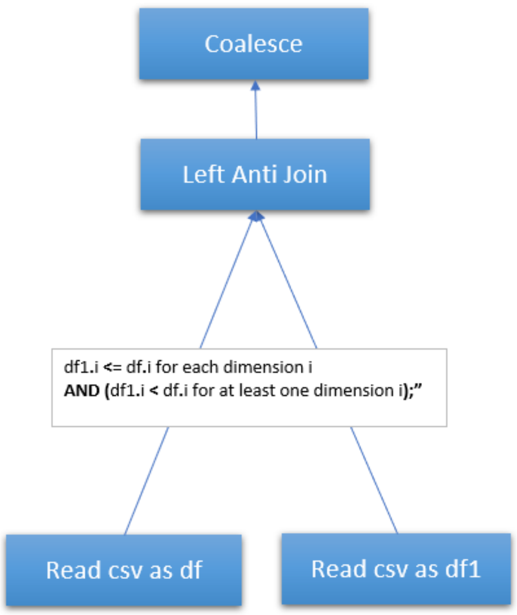
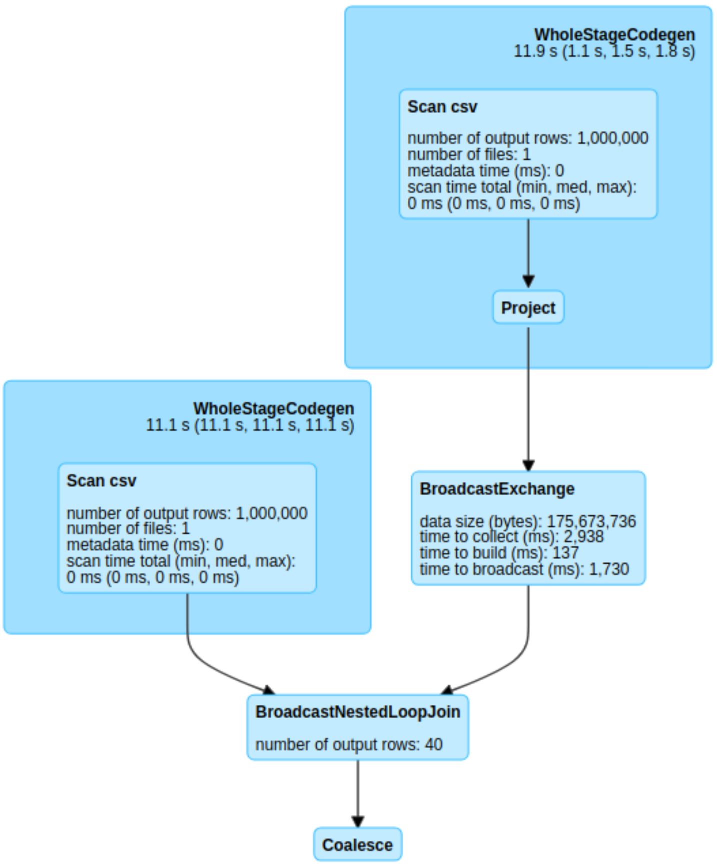

Coalesce
Left Anti Join
df1.i <= df.i for each dimension i
AND (df1.i < df.i for at least one dimension i);"
Read csv as df
Read csv as df1

Figure 13 DAG of post-optimization Nested SQL

The final DAG Spark creates is a physical representation of the optimized plan.


```
WholeStageCodegen
11.9 s (1.1 s, 1.5 s, 1.8 s)

Scan csv
number of output rows: 1,000,000
number of files: 1
metadata time (ms): 0
scan time total (min, med, max):
0 ms (0 ms, 0 ms, 0 ms)

Project

WholeStageCodegen
11.1 s (11.1 s, 11.1 s, 11.1 s)

Scan csv
number of output rows: 1,000,000
number of files: 1
metadata time (ms): 0
scan time total (min, med, max):
0 ms (0 ms, 0 ms, 0 ms)

BroadcastExchange
data size (bytes): 175,673,736
time to collect (ms): 2,938
time to build (ms): 137
time to broadcast (ms): 1,730

BroadcastNestedLoopJoin
number of output rows: 40

Coalesce
```

Figure 14 Physical representation of Nested SQL DAG

4.2.3 Grid Partitioning Algorithm

This algorithm aims to reduce the points collected in main memory. It projects the points of the dataset into equally sized cells. Each cell may contain 0 or more points of the dataset.

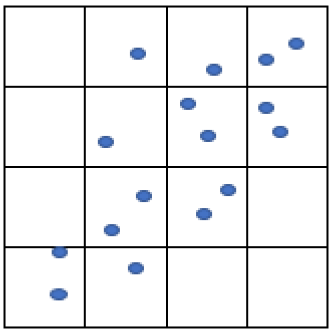


Figure 15 Dataset separated into cells

The cells are then examined and those that certainly cannot contain skyline points are eliminated. For the rest of the cells each partition's skylines are calculated in parallel and their results are merged in main memory when the final skyline points are calculated.

Grid formation

For the formation of the grid, the min and max values of every dimension of the database are necessary. The programmer sets the number of divisions each dimension should have. The matrix is then formed by calculating the boundaries of each cell for each dimension. A point of the dataset belongs to a cell when the value of each of its dimensions lied inside the cell's boundaries.

```
Algorithm gridMatrixCreation
Input: $D$: a multidimensional database
       $divisionType$: the number of divisions each dimension should have
Output: $boundaries$: a matrix containing the boundaries of each dimension
$min_i \coloneqq D_i.min,\ \ max_i \coloneqq D_i.max$ for each $D'$s dimension $i$
foreach dimension $i$ of $D$
    $interval \coloneqq (max_i - min_i)/divisionType$
    $boundaries_{i0} \coloneqq min_i$
    $boundaries_{ij} \coloneqq boundaries_{i,j-1} + interval$ for every j until $boundaries_{ij} == max_i$
return $boundaries$
```

Cell elimination

For a cell to be eliminated, it should be certain that another cell's points are dominating one-by-one all its points. In order to avoid further calculations and memory usage, the elimination process takes into consideration only the cells' index which reveals their relative position. For example, if the dataset has three dimensions, $cell_{1,3,1}$ is the first cell in the dimension-1, third in dimension-2 and first in dimension-3.

Based on the Skyline definition, it can be derived that:

Corollary: A cell *a* with index $x_1, x_2, \dots, x_n$ where n is the number of dimensions, cannot contain skyline points if and only if there exists a cell b with index $y_1, y_2, \dots, y_n$ where $y_i < x_i$ for each dimension i and b is non-empty.

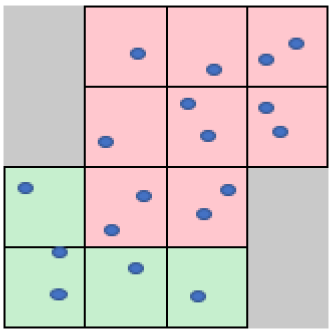


Figure 16 Dominated cells in Grid Partitioning

Following the corollary, an algorithm is constructed that takes as an argument the index of all the non-empty cells and returns the eliminated cells. The algorithm contains a volume of data small enough to be executed efficiently in the driver's machine and avoid network and I/O cost.

```
Algorithm getEliminatedCells
Input: $C$: an array of the index value of every non-empty cell $(e.g. index = (1,0,3)$
Output: $dominatedCells$: an array containing the indexes of the dominated cells
$dominatedCells \coloneqq \{\emptyset\}$
foreach $index \in C$
    $if\ \exists\ index2 \in C$ such that $index2.i < index.i$ for each dimension $i$:
        $dominatedCells.add(index)$
return $dominatedCells$
```

After the eliminated cells are extracted, they are sent to the nodes and the points each node contains are filtered not to be contained in those cells. Using the remaining nodes, the skyline nodes are calculated, at first in parallel for each node, and then in main memory.

```
**Algorithm** sparkGridPartitioning
**Input:** $D$ a multidimensional database
**Output:** $S$ a set of skyline points
$datasetRDD \coloneqq parallelize\ D\ into\ partitions$
$boundaries \coloneqq gridMatrixCreation(D, numberOfDivisions)$
do in parallel: append to each point of $datasetRDD$ its cell index
$neCells \coloneqq$ collect all the distinct cell indexes found in $datasetRDD$
$eliminatedCells \coloneqq getEliminatedCells(neCells)$
$DatasetRDD.filterInParallel(delete\ points\ with\ cell\ index\ in\ eliminatedCells)$
**foreach** $partition$ in $datasetRDD$ **do in parallel**
    $partition \coloneqq getSkylinePoints(partition)$
$datasetMain \coloneqq collect\_to\_main\_memory(datasetRDD)$
$S \coloneqq getSkylinePoints(datasetMain)$
```

Spark implementation

Once the source file is read and parallelized, the programmer sets the number of divisions each dimension should have. Then the division boundaries of each dimension are calculated.

```
val divisionType=3
val cellGrid = new CellGrid(rdd, divisionType)
val divisionBoundaries = cellGrid.getDivisionBoundaries()
```

Using the divisions' boundaries, each node calculates the cells their elements belong to. Then all the not empty cells are collected from this RDD object and the dominated cells are calculated.

```
val zippedRdd = addLocalDivisionPoints(rdd)
    .map(x => x
   .map(y => ( y._1._1, y._1._2, gridCalculation.getDimensionGridCell(y))))
    .map(x => (x, x.map(y => y._3)))
    .map(x => x.swap)
    .map(x => (x._1.toList, x._2))
    .map(x => (x._1, x._2.toIterable))
def addLocalDivisionPoints(rdd:RDD[Array[Double]])  =
  {
    rdd.map(x => x.zipWithIndex)
      .map(x => x.map(y => (y, divisionPointsB.value(y._2))))
  }
val necells = zippedRdd.map(x => x._1).distinct().collect()
val dominatedCells = getDominatedCells(necells.toList)
```

Each node filters its tuples and returns only the points that are not a part of the dominated cells.

```
val filteredCellsRDD = filterDominated(zippedRdd)
    .map(x => x._2.map(y => y._1).toArray)
```

Then, for each partition, the algorithm calculates the local skyline points using the SFS Skyline calculation technique which is described in the next paragraph. Finally, the resulted local skylines are collected into the driver and the global skyline points are calculated and extracted.

The execution plan Spark creates consists of 4 core jobs. The first two, return the min and max values of the dataset and are executed for each dimension during the creation of the grid boundaries. The next job is terminated once the distinct non-empty cells are extracted from the RDD object and written in main memory. The last job terminates when each local skyline tuple is collected and is used for the final skyline calculation. Minor jobs are not mentioned because they do not affect significantly the efficiency of the algorithm.

4.3 Skyline calculation techniques

Regardless of whether the skyline points are calculated for each node in parallel, or in the driver, the inputs and outputs of the skyline calculation are of the same object type. When performing a node skyline calculation, the input is the points of a single partition of an RDD object. The output is stored in the same partition before being merged with every partition's output in the driver. In main memory, the input is a collection of local skyline points and after the calculation the output is written in file. Two approaches for the skyline calculation were examined. The first, *simpleSkylineCalculation* compares each point of the input with the rest. If a domination condition exists during the comparison, the dominated point is deleted from the dataset. The second *SFSkylineComputation*, is the implementation of *Ilaria Bartolini*'s algorithm [29];It first sorts the dataset in ascending order according to a monotone preference function. The first point is inserted to a candidate list and the components of the list are compared with the rest of points. If a point dominates one or more points of the list, that points are deleted. If the point is not dominated by any point of the list, it is inserted in the list.

During the first experiments, it became appreciable that *SFSkylineComputation* was a lot more efficient than the *simpleSkylineCalculation* and was for that reason integrated in each of the thesis' algorithms.

```
Algorithm simpleSkylineCalculation
Input: $D$ a set of points
Output: $S$ a set of skyline points
$S := D$
foreach point $p_1$ in $S$ \\loop1
    foreach point $p_2 \neq p_1$ \\loop2
        if $p_1.dominates(p_2)$
            remove $p_2$ from $S$
        else if $p_2.dominates(p_1)$
            remove $p_1$ from $S$
            exit loop2
return $S$
```

```
Algorithm SFSkylineCalculation
Input: $D$ a set of points
Output: $S$ a set of skyline points
Add score on each tuple of $D$
Sort $D$ according to their score
Initiate $S := D(0)$
foreach point $p_1$ in $D$ except $D(0)$ \\loop1
    $toBeAdded := True$
    foreach $p_2$ in $S$ \\loop2
        if $p_1.dominates(p_2)$
            remove $p_2$ from $S$
        else if $p_2.dominates(p_1)$
            $toBeAdded := False$
            exit loop2
    if $toBeAdded == True$
        $S.add(p_1)$
return $S$
```

```
Algorithm dominates
Input: $p_1, p_2$ points
Output: $booleanValue$ True or False
$booleanValue := (p_1.i \leq p_2.i$ for each dimension $i)$ AND
                 $(p_1.i < p_2.i$ for at least one dimension $i)$
Return $booleanValue$
```

4.4 Optimization of the algorithms

The Apache Spark architecture and its components' functionalities have a key role to the effectiveness of the algorithms. It is highly important that the algorithms are programmed so that they take advantage of the Spark capabilities in an optimal manner.

As aforementioned in chapter 3, a major cause of delays in a Spark program is the shuffling operators. They result to highly expensive data movements and replications across different nodes and should be avoided whenever possible. The cost of each shuffling operator varies, with *groupByKey* being the most expensive. For that reason, in all three algorithm implementations, the *-byKey* algorithms are avoided. For example, in the grid partitioning algorithm the tuples are not grouped or aggregated according to their grid cell, but each tuple's cell is appended to the RDD element. The non-empty cells are extracted from the RDD object by a *distinct* operator which causes minor shuffling between the nodes.

The algorithms take advantage of the Broadcast functionality of Spark. Some of the algorithms' operators require data that are stored in the driver's memory. Spark requires those data to be copied in each node in order to participate in an RDD function's result. When the data are broadcasted instead, they are stored in each machine rather than each node, and the redundant memory allocation is avoided. For that reason, variables necessary for some parallel calculations like the dominated grid cells and the dimension's boundaries are broadcasted instead of copied.

The grid partitioning algorithm, being the most complex of the three, requires a sequence of actions and transformation to be executed in the same RDD objects. By default, when Spark recognizes an action, it plans a job that contains all the transformations appearing between the RDD object's creation and action. When those transformations appear again later in the program, Spark re-executes them resulting to unnecessary calculations. The grid partitioning algorithm uses the *persist* function when suitable, to maintain the transformed object after an action is performed on it.

One of the costliest phases of all the algorithms is the collection and skyline calculation executed in the driver. By using the simple skyline calculation, every tuple is compared with every other tuple until a domination relation between them appears. In order to collect and compute the final skyline points efficiently and not overload the driver's memory, an adjustment of the SFS Skyline method is used. The first tuple collected in the driver, is inserted into an $\langle ArrayBuffer \rangle$ object. Each new collected tuple is compared only with this object which is updated when a skyline point is detected or dominated. The elements of the Buffered Array, after all the points are examined, form the final skyline points of the dataset. This method achieves to avoid redundant comparisons between the tuples.

5 Performance evaluation using a multi-core machine and a cluster of machines

During this chapter, the results of experiments on the three algorithmic approaches are presented and discussed. Each approach offers different pros and cons and it is difficult to weight their efficiency without testing them in various environments using different input sources.

5.1 Algorithms strengths and weaknesses

5.1.1 ALS

Advantages

The All Local Skyline algorithm implements the simplest approach to distributed skyline computation. By relying on Spark's default partitioning method, the data do not need to be accessed before the computation of the local skyline points. It is also highly likely that each partition follows the same distribution with the whole dataset, which means that the skyline points retrieved from every partition are at a certain level relative to the final skyline points. In addition, costly shuffling operations between partitions are not present. Likewise, the collection of those points and the driver's skyline computation is not based on complex procedures.

Disadvantages

The main disadvantage of ALS is that the local skyline points are calculated for each partition without performing a prior mass elimination of points, as opposed to the grid partitioning approach. Moreover, scaling the volume of the local skyline points that are finally collected in main memory affects dramatically the skyline computation time. While the computational cost of distributed calculations can be easily reduced by appending more nodes in the cluster, the hardware of the central machine is restrictive to high scalability.

5.1.2 Nested SQL

Advantages

This approach relies heavily on the optimization techniques of SparkSQL's Catalyst optimizer. While single-machine SQL commands perform redundant non-scalable calculations, SparkSQL is designed to efficiently use the cluster's distribution and optimize the queries' plan according to it.

Disadvantages

The Catalyst optimizer seeks to optimize the application's queries for an effective, distributed execution. The query itself though is not always capable of transforming the problem's solution in the most detailed and efficient approach, unlike the RDDs and the agility they offer to the programmer. In other words, the optimization and execution of the algorithm's plan is a black box to the programmer, who is provided with limited methods of affecting it.

5.1.3 Grid Partitioning

Advantages

Grid partitioning is able to perform early eliminations of tuples the volume of which depends highly on the distribution of the dataset. While this adds more tasks to be executed, the calculations for grid elimination are applied in a small volume of data in the driver, without network delays. By filtering the eliminated cells from the RDD object the local skyline calculations are executed more quickly in each partition and finally the tuples returned for the main memory skyline calculation are reduced as well.

Disadvantages

It is controvertible whether the addition of the grid elimination tasks and the filtering methods to the execution plan reduces the volume data in such level that the performance of the algorithm increases. In addition, the programmer must optimally adjust the number of divisions each dimension should have according to the volume and distribution of the dataset, the characteristics of the cluster and the hardware capabilities of the driver.

5.2 Dataset

To evaluate the efficiency of the algorithms, a collection of datasets is created. Each dataset contains randomly generated float numbers following a specific distribution. Figures

17-20 display the distributions used for the algorithms' evaluation and the skyline tuples each produces.

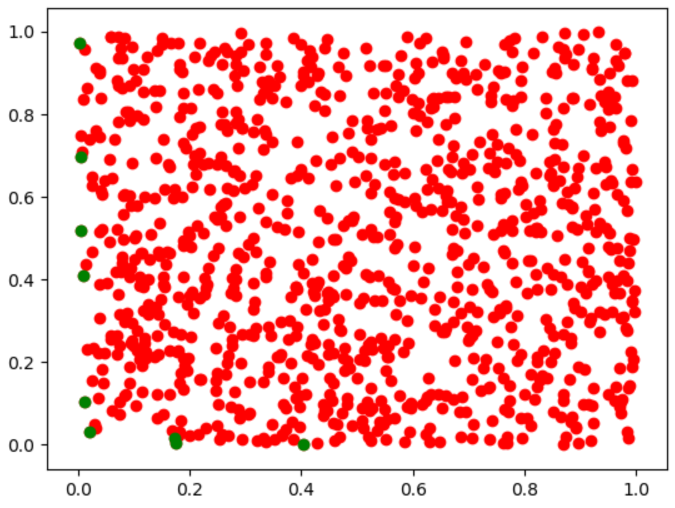


Figure 17 skyline points in a uniform dataset

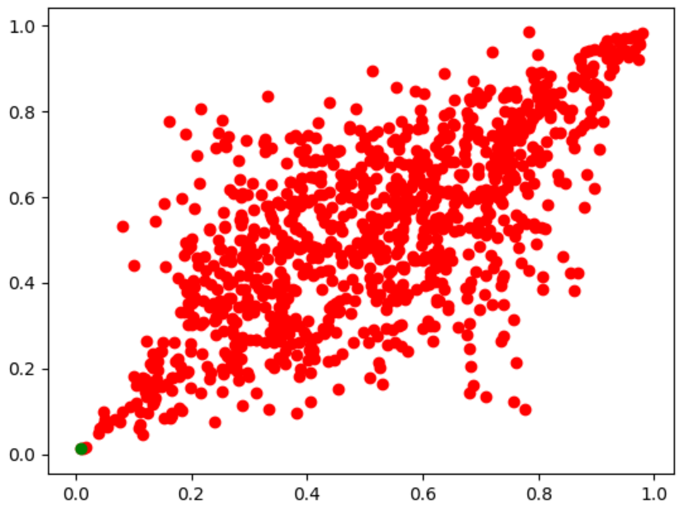


Figure 18 skyline points in a correlated dataset

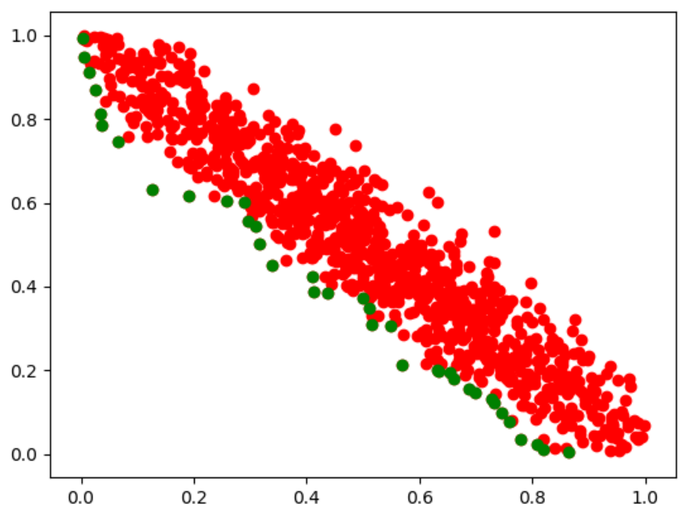


Figure 19 skyline points in an anticorrelated dataset

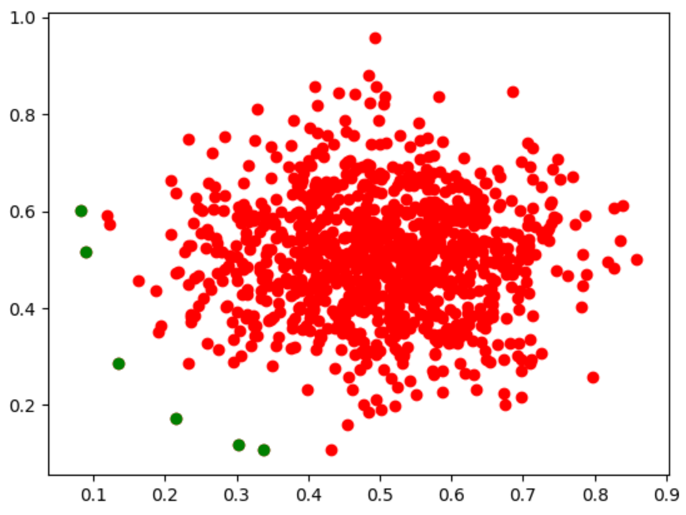


Figure 20 skyline points in a gaussian dataset

As the figures indicate, the distribution of a database highly affects the number of Skyline points it contains. While the majority of the tuples in correlated data are dominated, in anticorrelated data a significant number of tuples is skyline points. An efficient skyline calculation algorithm should perform well on any type of distribution without prior knowledge of this type.

Since the thesis' algorithms are implemented in Spark, which is a platform for parallel and distributed data processing, the scalability of the algorithms is evaluated by applying them on databases of various sizes, from 1.000.000 to 100.000.000 tuples.

5.3 Performance on Standalone mode

The algorithms are firstly evaluated on a single machine, with Intel Core I7 having 4 cores and 8 threads. The data examined have 100.000 3-dimension (small), 500.000 4-dimension (medium) and 1.000.000 5-dimension (large) tuples.

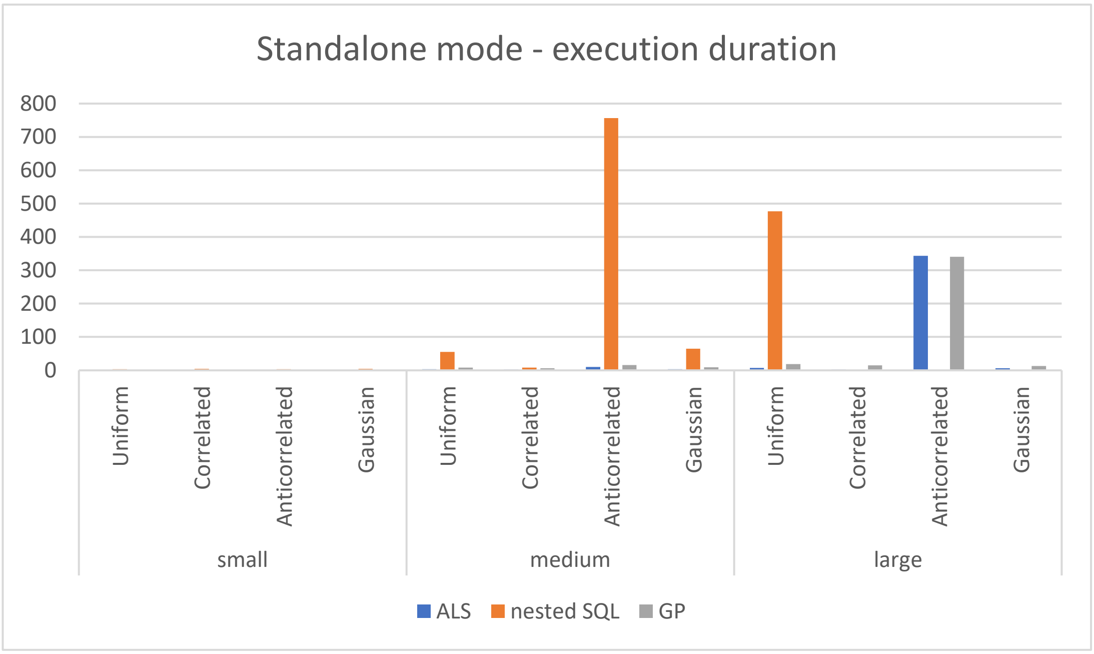


Figure 21 Execution duration in standalone mode

The Nested SQL algorithm is always slower than the other two algorithms. It should be noted that the algorithm's results were able to be retrieved only for small and medium data and exclusively large uniform data. When excluding Nested SQL, the graph becomes more interesting regarding the efficiency of ALS and Grid Partitioning.

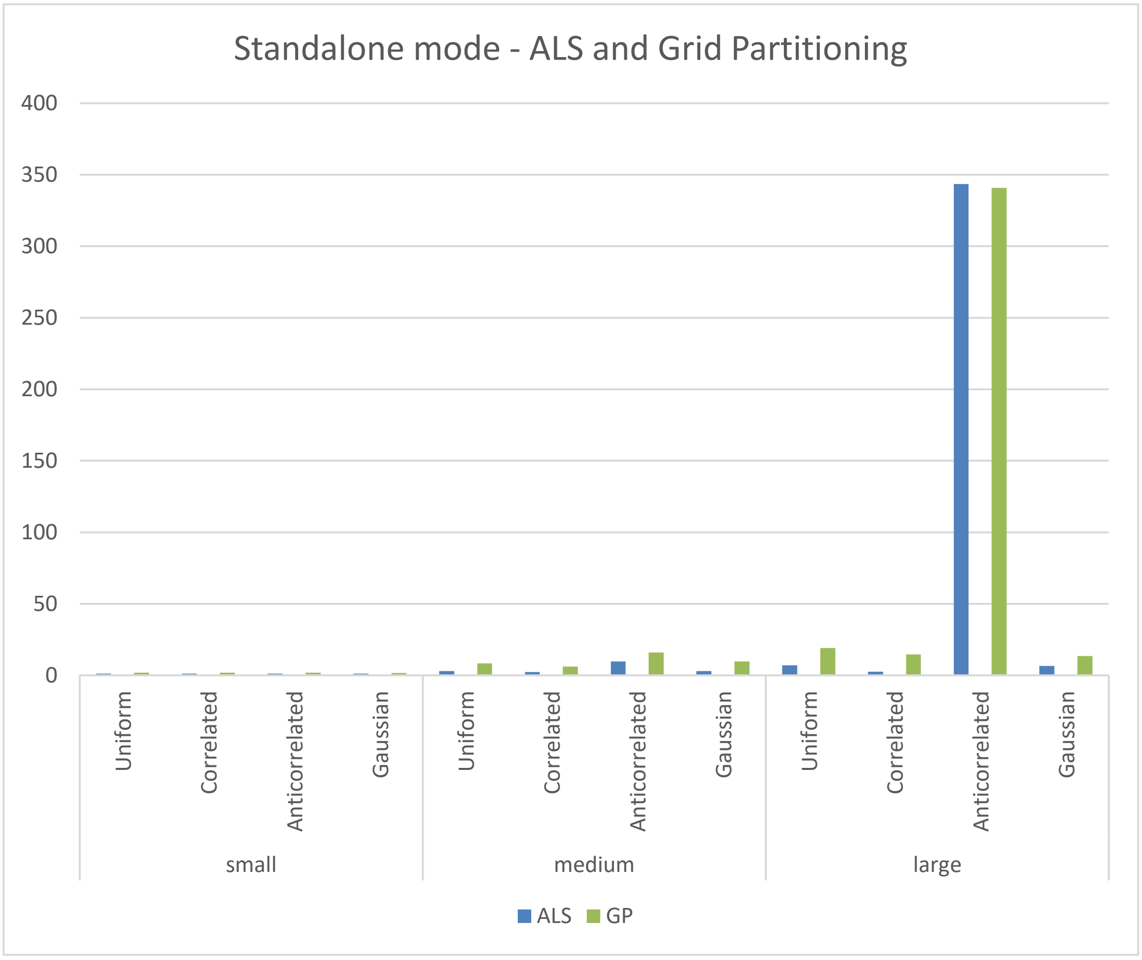


Figure 22 Execution duration of ALS and Grid Partitioning in standalone mode

The figure shows that the difference between the two algorithms in standalone mode is small at all cases. ALS is always slightly better with the exception of large anticorrelated data. Moreover, anticorrelated data require more time to be processed since they contain more skyline tuples.

5.4 Performance on a Hadoop cluster

The algorithms are executed using Hadoop's Yarn with 1.000.000 and 10.000.000 tuples, and 4 and 8 executors. The datasets used are retrieved from the Hadoop Distributed Filesystem, and consist of 3-dimensional tuples of each distribution type. The local skyline calculation and the total processing time can be compared by observing figures 23-26. Nested SQL algorithm was not able to return results due to time-out errors.

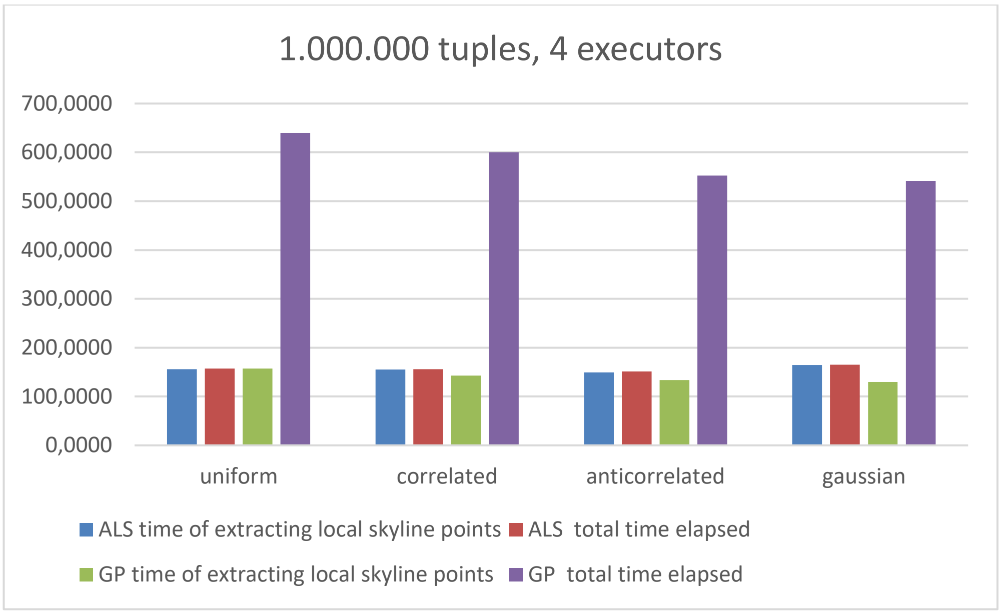


Figure 23 Algorithms' performance in 1000000 tuples, 4 executors

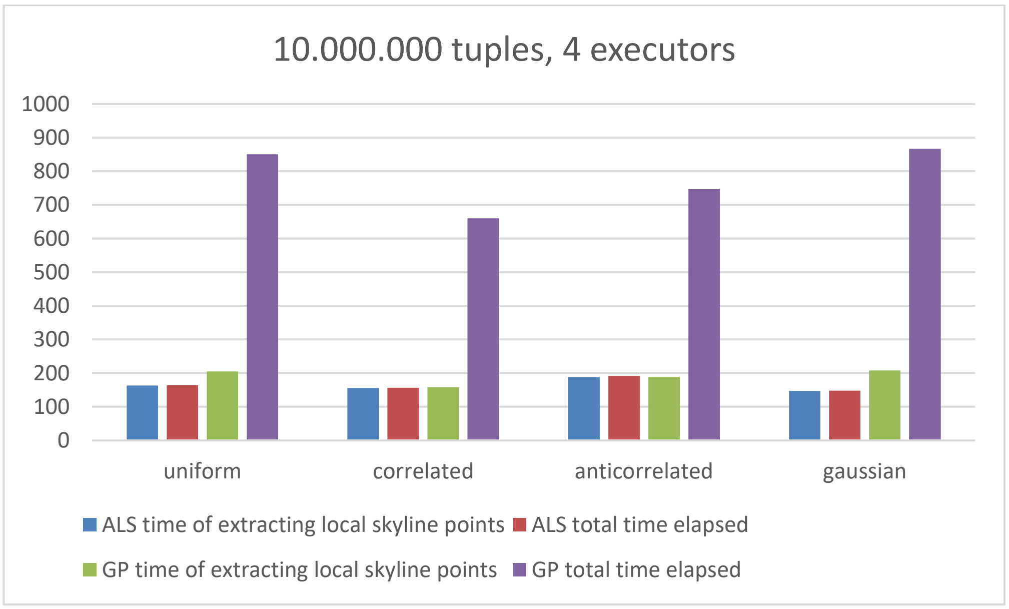


Figure 24 Algorithms' performance in 10000000 tuples, 4 executors

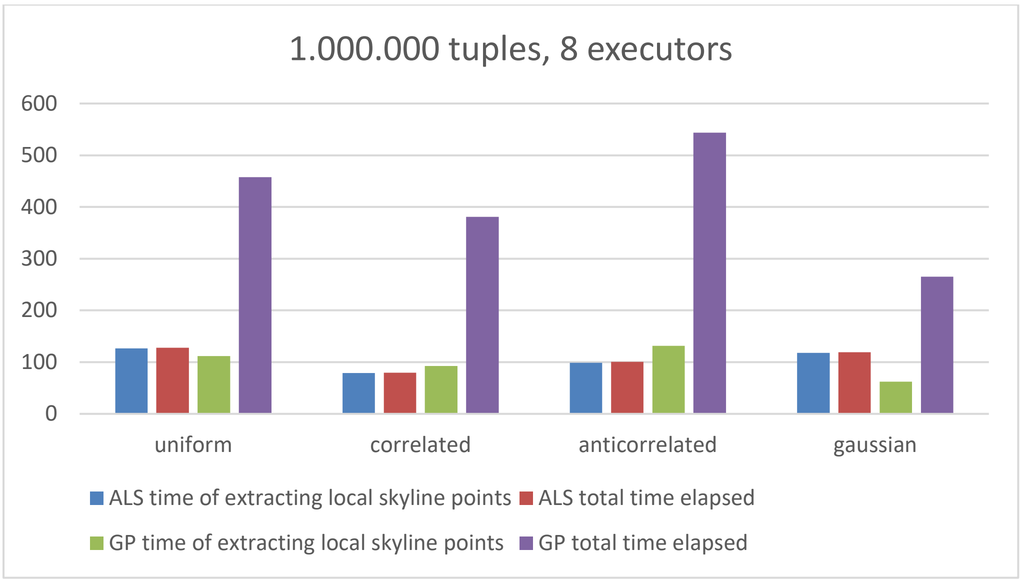


Figure 25 Algorithms’ performance in 100000 tuples, 8 executors

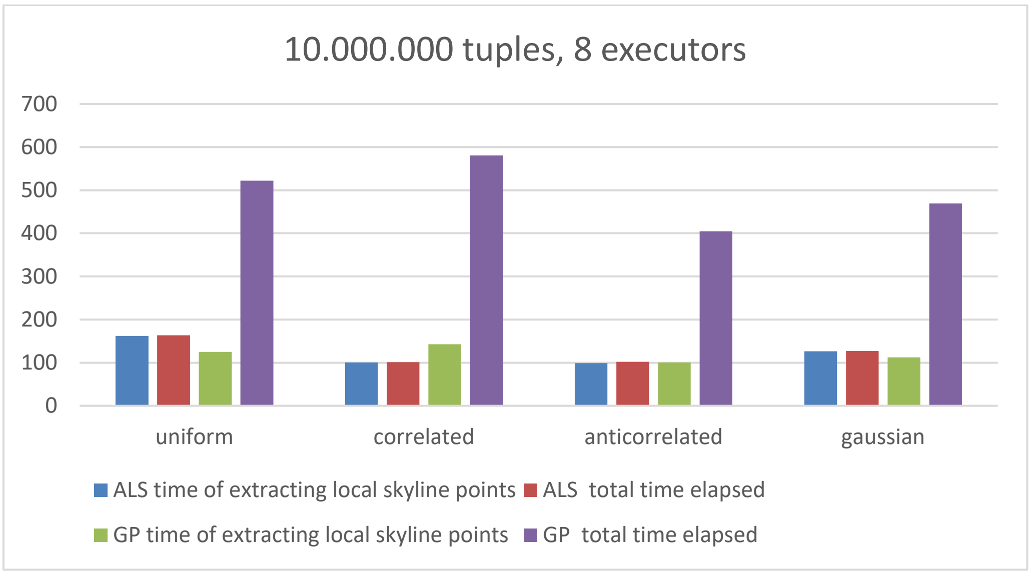


Figure 26 Algorithms’ performance in 10000000 tuples, 4 executors

Regarding the distribution of the datasets, it is noticed that under these circumstances it does not affect noticeably the total execution of the algorithms.

An easy observation is that ALS is significantly more efficient in both cases. Although Grid Partitioning offers sophisticated ways of early elimination of tuples and calculates the local skylines in a slightly shorter period, this is achieved with a serious cost. Figure 27 shows in detail on which stages, during the execution, Grid Partitioning delays the execution time.

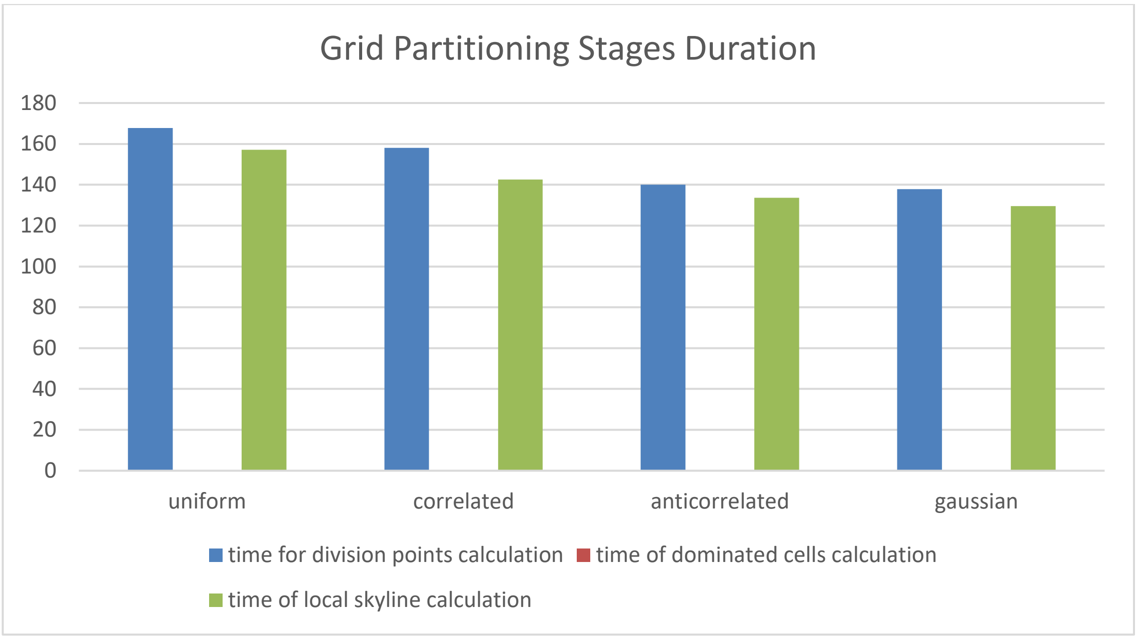


Figure 27 Grid Partitioning Stages Duration

The figure shows that although the local skyline calculation in Grid Partitioning is quicker than in ALS, the cost of calculating each dimension’s boundaries is very high. Indeed, for this calculation, costly min and max actions are performed for each tuple’s dimension, requiring coordination between the nodes. On the other hand, the driver calculates the dominated cells instantly, confirming that its memory and executors are adequate for this task. Another costly procedure of Grid Partitioning is the collection of the non-empty cells from the RDD object to driver’s memory.

Next, the scalability of the algorithms is examined. They are executed using 8 executors in 1.000.000, 10.000.000, 50.000.000 and 100.000.000 uniform distributed tuples. The results are displayed below.

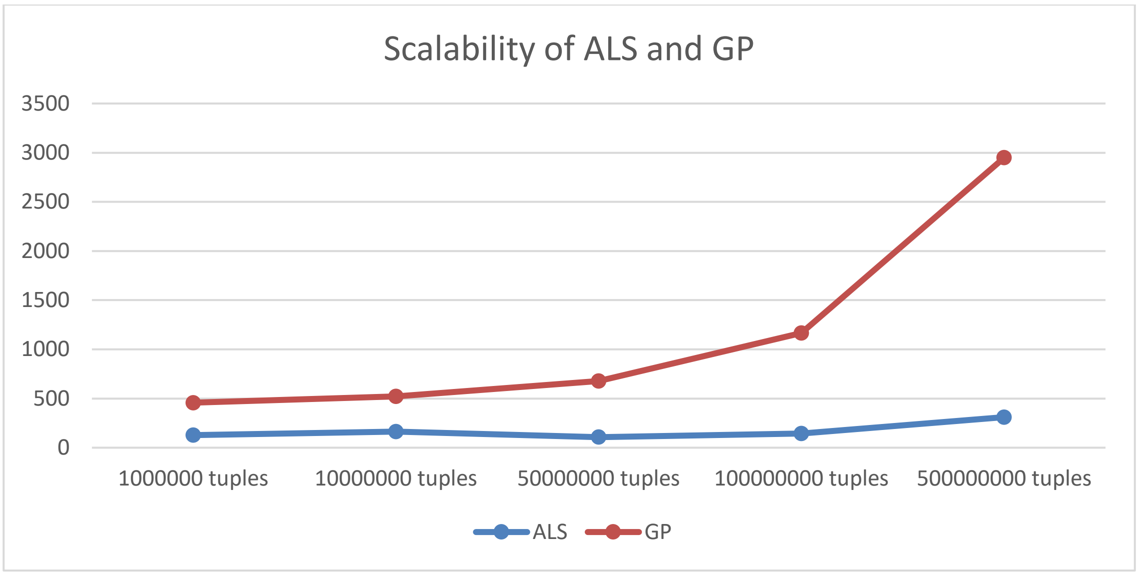


Figure 28 Scalability of ALS and Grid Partitioning

It is clear that the performance of the ALS algorithm, unlike Grid Partitioning, is almost not affected by the volume of the input data, despite the fact that more skylines tuples are returned to the driver program.

The overall results were in favor of the ALS algorithm and along with the insights gained from multiple attempts before reaching to the final structure of the algorithms can indicate some inferences:

- Apache Spark's architecture allows pipelined calculations performed on each node to be executed in a fairly short time. This can be noticed by the minor difference of ALS and Grid Partitioning when computing the local skyline calculations, although Grid Partitioning's tuples at that point are significantly reduced.
- Multiple actions add a cost to the execution of the algorithms that should not be neglected. The calculation of the grid cells almost doubles the duration of the execution due the min and max values extracted for each dimension of the dataset. The collection of the non-empty from the RDD object to the driver cells is also very time consuming.
- Persist, unpersist and broadcast functions increased the efficiency of the algorithms.
- Shuffling operators, on the other hand, increase the execution time and should only be used if completely necessary.

- SparkSQL’s optimizer was not able to adapt efficiently to the given query. While the SQL approach was able to handle adequately small-sized datasets, it failed during the scaling of the data, due to its expensive broadcast operations.

6 Conclusions and future work

This chapter presents the conclusions derived after the completion of the thesis and offers ideas for potential future work.

The objective of this dissertation was the implementation and evaluation of efficient, scalable Skyline calculation algorithms using the Apache Spark framework and the Scala programming language. It provided motivation for the basic acquisition of the Scala programming environment, the Skyline calculation problem and more importantly the architecture and programming environment of the constantly evolving Apache Spark framework. The algorithms designed where based on propositions from related scientific literature and where adapted to Apache Spark's architecture.

The three algorithmic approaches implemented are ALS, Nested SQL and Grid Partitioning. In ALS, local skyline tuples are calculated for each node which are then merged in main memory to result to the final skyline tuples. In nested SQL, the result is retrieved by performing an SQL query, which is automatically optimized by Spark's Catalyst optimizer. The Grid Partitioning algorithm separates the data space into cells and performs early elimination of tuples that are contained in dominated cells. Those algorithms where executed in both a single machine and a Hadoop environment using multiple executors. The results have shown that ALS is the most suitable approach among the three for Skyline calculation on Spark, due to the simplicity of the execution's DAG. Nested SQL has proven inadequate on large-scale datasets while the Grid Partitioning algorithm's cell elimination cost was not counterbalanced by the minor cost reduction of the local Skyline calculation. Choosing to use alterations of the SFS skyline calculation rather than the basic skyline calculation, reduced significantly the driver's and partitions' calculation time, thus is proven to be suitable for Skyline querying in Spark.

Overall, the experiments have shown that designing an efficient Spark Skyline calculation algorithm cannot entirely rely on the propositions of the existing distributed Skyline calculation research papers because of Spark's distinct architecture. The lack of communication between the worker nodes compels for mostly pipelined workflows and minimized data exchanges among the workers, and between the nodes and the driver.

Besides the three implemented algorithms, there are many other algorithmic approaches that can be adjusted in the Spark platform and be evaluated. Angle-based Space

Partitioning, Kian-Lee Tan's [4] approach of transforming the tuples into bitmaps, are just a few examples of approaches potentially useful for Spark.

The SparkSQL abstraction of Spark, could also benefit by the addition of a Skyline query operator that is programmed to return rapid, efficient and scalable results, instead of relying to the restrictive SQL commands. Since Spark supports streaming data, skyline algorithms can also be designed for data of such type, allowing the application of Skyline calculation to real world problems just like the search of a mobile phone between multiple different stores .

In general, distributed Skyline calculation literature, which until today consists mostly of approaches based on peer-to-peer environments, can be enriched with publications concerning efficient Skyline calculation on Apache Spark, and other master/worker-based distributed systems.

Bibliography

[1] S. Borzsony, D. Kossmann and K. Stocker, "The Skyline operator," Proceedings 17th International Conference on Data Engineering, Heidelberg, Germany, 2001, pp. 421-430. DOI=https://doi.org/10.1109/ICDE.2001.914855

[2] H. T. Kung, F. Luccio, and F. P. Preparata. 1975. On Finding the Maxima of a Set of Vectors. J. ACM 22, 4 (October 1975), 469-476. DOI=http://dx.doi.org/10.1145/321906.321910

[3] Christian Buchta, On the average number of maxima in a set of vectors, Information Processing Letters, Volume 33, Issue 2, 1989, Pages 63-65. DOI=https://doi.org/10.1016/0020-0190(89)90156-7

[4] Tan, Kian-Lee & Eng, Pin-Kwang & Chin Ooi, Beng. (2001). Efficient Progressive Skyline Computation. 301-310

[5] Kossmann, Donald & Ramsak, Frank & Rost, Steffen. (2002). Shooting Stars in the Sky. 275-286. DOI=http://dx.doi.org/10.1016/B978-155860869-6/50032-9

[6] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. 2003. An optimal and progressive algorithm for skyline queries. In Proceedings of the 2003 ACM SIGMOD international conference on Management of data (SIGMOD '03). ACM, New York, NY, USA, 467-478. DOI=https://doi.org/10.1145/872757.872814

[7] Chomicki, Jan & Godfrey, Parke & Gryz, Jarek & Liang, Dongming. (2003). Skyline with presorting. Proceedings - International Conference on Data Engineering. 717- 719. DOI=http://dx.doi.org/10.1109/ICDE.2003.1260846

[8] Xuemin Lin, Yidong Yuan, Wei Wang, and Hongjun Lu. 2005. Stabbing the Sky: Efficient Skyline Computation over Sliding Windows. In Proceedings of the 21st International Conference on Data Engineering (ICDE '05). IEEE Computer Society, Washington, DC, USA, 502-513. DOI=https://doi.org/10.1109/ICDE.2005.137

[9] Mikhail J. Atallah and Yinian Qi. 2009. Computing all skyline probabilities for uncertain data. In Proceedings of the twenty-eighth ACM SIGMOD-SIGACT-

SIGART symposium on Principles of database systems (PODS '09). ACM, New York, NY, USA, 279-287. DOI=https://doi.org/10.1145/1559795.1559837

[10] Jian Pei, Bin Jiang, Xuemin Lin, and Yidong Yuan. 2007. Probabilistic skylines on uncertain data. In Proceedings of the 33rd international conference on Very large data bases (VLDB '07). VLDB Endowment 15-26.

[11] X. Ding and H. Jin, "Efficient and Progressive Algorithms for Distributed Skyline Queries over Uncertain Data," in IEEE Transactions on Knowledge and Data Engineering, vol. 24, no. 8, pp. 1448-1462, Aug. 2012. doi= http://dx.doi.org/10.1109/TKDE.2011.77

[12] Hose, Katja & Vlachou, Akrivi. (2012). A survey of skyline processing in highly distributed environments. The Vldb Journal - VLDB. 21. 1-26. DOI=http://dx.doi.org/10.1007/s00778-011-0246-6

[13] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. 1989. Programming languages for distributed computing systems. ACM Comput. Surv. 21, 3 (September 1989), 261-322. DOI=http://dx.doi.org/10.1145/72551.72552

[14] Satyanarayanan, Mahadev. (1995). A Survey of Distributed File Systems. Annual Review of Computer Science. 4. DOI=http://dx.doi.org/10.1146/annurev.cs.04.060190.000445

[15] HDFS Architecture Guide (source: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html)

[16] L. Zhu, Y. Tao and S. Zhou, "Distributed Skyline Retrieval with Low Bandwidth Consumption," in IEEE Transactions on Knowledge and Data Engineering, vol. 21, no. 3, pp. 384-400, March 2009. DOI=http://dx.doi.org/10.1109/TKDE.2008.142

[17] Huang, Zhiyong & Jensen, Christian & Lu, Hua & Chin Ooi, Beng. (2006). Skyline Queries Against Mobile Lightweight Devices in MANETs. Proceedings - International Conference on Data Engineering. 2006. 66- 66. DOI=http://dx.doi.org/10.1109/ICDE.2006.142

[18] Vlachou, C. Doulkeridis, Y. Kotidis and M. Vazirgiannis, "SKYPEER: Efficient Subspace Skyline Computation over Distributed Data," 2007 IEEE 23rd International Conference on Data Engineering, Istanbul, 2007, pp. 416-425. DOI=http://dx.doi.org/10.1109/ICDE.2007.367887

[19] Cui, H. Lu, Q. Xu, L. Chen, Y. Dai and Y. Zhou, "Parallel Distributed Processing of Constrained Skyline Queries by Filtering," 2008 IEEE 24th

International Conference on Data Engineering, Cancun, 2008, pp. 546-555. DOI=http://dx.doi.org/10.1109/ICDE.2008.4497463

[20] Wu, Ping & Zhang, Caijie & Feng, Ying & Y. Zhao, Ben & Agrawal, Divyakant & El Abbadi, Amr. (2006). Parallelizing Skyline Queries for Scalable Distribution. 3896. 112-130. DOI=http://dx.doi.org/10.1007/11687238_10

[21] Vlachou, Akrivi & Doulkeridis, Christos & Kotidis, Yannis. (2008). Angle-based space partitioning for efficient parallel skyline computation. 227-238. DOI=http://dx.doi.org/10.1145/1376616.1376642

[22] S. Wang, B. C. Ooi, A. K. H. Tung and L. Xu, "Efficient Skyline Query Processing on Peer-to-Peer Networks," 2007 IEEE 23rd International Conference on Data Engineering, Istanbul, 2007, pp. 1126-1135. DOI=http://dx.doi.org/10.1109/ICDE.2007.36897

[23] Mullesgaard, K, Pederseny, JL, Lu, H & Zhou, Y 2014, Efficient Skyline Computation in MapReduce. in Proc. 17th International Conference on Extending Database Technology (EDBT), Athens, Greece, March 24-28, 2014.. EDBT, pp. 37-48. DOI=https://doi.org/10.5441/002/edbt.2014.05

[24] Hagedorn, Stefan & Sattler, Kai-Uwe & Gertz, Michael. (2015). A Framework for Scalable Correlation of Spatio-temporal Event Data. DOI=http://dx.doi.org/10.1007/978-3-319-20424-6_2

[25] Konstantinos Paparidis (2016), Parallel and Distributed Skyline Computation on Spark System, source=http://ikee.lib.auth.gr/record/285311/?ln=el

[26] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. Commun. ACM 59, 11 (October 2016), 56-65. DOI=https://doi.org/10.1145/2934664

[27] RDD programming Guide, source=https://spark.apache.org/docs/latest/rdd-programming-guide.html

[28] Jacek Laskowski, Mastering Apache Spark, source=https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/

[29] Bartolini, I., Ciaccia, P., & Patella, M. (2008). Efficient sort-based skyline evaluation. ACM Trans. Database Syst., 33, 31:1-31:49.

[30] Schröder, Bernd S. W. (2003). Ordered Sets: An Introduction. Birkhäuser Boston 2003. DOI= https://doi.org/10.1007/978-1-4612-0053-6

[31] Zaharia, Matei & Chowdhury, Mosharaf & J. Franklin, Michael & Shenker, Scott & Stoica, Ion. (2010). Spark: Cluster Computing with Working Sets. Proceedings of the 2nd USENIX conference on Hot topics in cloud computing. 10. 10-10.

Appendix

Figures table

6.1 Code

skylineMain.scala

```
object skylineMain {
  def main(args: Array[String]) {
    val conf = new SparkConf().setAppName("skylineCalculator
    val sc = new SparkContext(conf)
    val now = System.nanoTime
    val algorithm = args(1)

    algorithm match {
      case "nestedSQL" => new nestedSQL(args(0), args(2).toDouble)
      case "gp" => new gridPartitioning(args(0), sc, args(2).toDouble)
      case "rddBasicSFS" => new rddBasicSFS(args(0), sc, args(2).toDouble)
      case _ => println("algorithm not yet implemented")
   }

    val timeElapsed = System.nanoTime - now
    println("total time elapsed: "+ timeElapsed.asInstanceOf[Double] / 1000000000.0)
  }
}
```

algorithms.ALS.scala

```
class ALS(inputPath: String, sc: SparkContext,  samplingRate: Double) extends Serializable {
  val inputingTime = System.nanoTime


  val rdd = sc.textFile(inputPath).sample(withReplacement = false, samplingRate)
  println("rdd created")
  println("number of tuples: "+1000000000*samplingRate)
  println("number of rdd's partitions:" + rdd.getNumPartitions)
  val rdd2 = rdd.map(x=>x.split(" ")).map(x => x.map( y => y.toDouble)).mapPartitions(SFSSkyline-
Calculation.addScoreAndCalculate)
  rdd2.persist()
  println("number of local skylines: "+rdd2.count())
  val localSkylinesTime = System.nanoTime
  println("time of extracting local skyline points:"+(localSkylinesTime-inputingTime).asIn-
stanceOf[Double] / 1000000000.0)
  var partitionSkylines = ArrayBuffer[Array[Double]]()
 rdd2.collect.foreach(x => SFSSkylineCalculation.calculatePartition(partitionSkylines, Itera-
tor(x))
  )
  println("skyline completed. total skylines:"+partitionSkylines.length)
  println("time of extracting final skylines:"+(System.nanoTime-localSkylinesTime).asIn-
stanceOf[Double] / 1000000000.0)
}
```

algorithms.gridPartitioning.scala

```
class gridPartitioning (inputPath: String, sc: SparkContext,  samplingRate: Double) extends Seri-
alizable {
  val rdd = sc.textFile(inputPath).sample(withReplacement = false,  samplingRate: Double)
    .map(x=>x.split(" "))
    .map(x => x.map( y => y.toDouble))
  println("number of tuples:"+1000000000*samplingRate)
  println("rdd created")
  println("number of rdd's partitions: " + rdd.getNumPartitions)

  val divisionType=5
  val beforeGridCalculation = System.nanoTime
  val partition = new Partition(rdd, divisionType)
  val divisionPoints = partition.getDivisionPoints()
  val afterGridCalculation = System.nanoTime
  println("time for division points calculation: "+(afterGridCalculation-beforeGridCalcula-
tion).asInstanceOf[Double] / 1000000000.0)
  val divisionPointsB = sc.broadcast(divisionPoints)
  val emptySet = List[Array[(Double, Int, Int)]]()
  val zippedRdd = addLocalDivisionPoints(rdd)
    .map(x => x.map(y => ( y._1._1, y._1._2, gridCalculation.getDimensionGridCell(y))))
    .map(x => (x, x.map(y => y._3)))
    .map(x => x.swap)
    .map(x => (x._1.toList, x._2))
    .map(x => (x._1, x._2.toIterable))
  zippedRdd.persist()
  rdd.unpersist()

  val necells = zippedRdd.map(x => x._1).distinct().collect()
  println("total number of cells:" + necells.length)
  val beforeCellElimination = System.nanoTime
  val dominatedCells = getDominatedCells(necells.toList)
val afterCellElimination = System.nanoTime

  println("number of dominated cells:" + dominatedCells.size)
  println("time of dominated cells calculation:"+(afterCellElimination-beforeCellElimina-
tion).asInstanceOf[Double] / 1000000000.0)

  val dominatedCellsB = sc.broadcast(dominatedCells)

  val filteredCellsRDD = filterDominated(zippedRdd)
    .map(x => x._2.map(y => y._1).toArray)
  filteredCellsRDD.persist()
  zippedRdd.unpersist()
  val nOfTuples = filteredCellsRDD.count()
  println("number of tuples after cell elimination:" + nOfTuples)
  val beforeLocalSkylines = System.nanoTime
  val skylinedRDD=filteredCellsRDD.mapPartitions(SFSSkylineCalculation.addScoreAndCalculate)
  val skylinedRDDwithIndex = addLocalDivisionPoints(skylinedRDD)
    .map(x => x.map(y => ( y._1._1, y._1._2, gridCalculation.getDimensionGridCell(y))))
    .map(x => (x, x.map(y => y._3)))
    .map(x => x.swap)
    .map(x => (x._1.toList, x._2))
    .map(x => (x._1, x._2.toIterable))
  skylinedRDDwithIndex.persist()
  filteredCellsRDD.unpersist()
  println("number of local skylines in all the partitions:"+skylinedRDDwithIndex.count())
  val afterLocalSkylines = System.nanoTime
  println("time of local skyline calculation:"+(afterLocalSkylines-beforeLocalSkylines).asIn-
stanceOf[Double] / 1000000000.0)
var partitionSkylines = ArrayBuffer[Array[Double]]()
  skylinedRDDwithIndex.map(x => x._2.map(y => y._1).toArray).collect.foreach(x=>SFSSkylineCalcu-
lation.calculatePartition(partitionSkylines,Iterator(x)))
  println("skyline completed. total skylines:"+partitionSkylines.length)
  println("time of extracting final skylines:"+(System.nanoTime-afterLocalSkylines).asIn-
stanceOf[Double] / 1000000000.0)
def filterDominated(rdd:RDD[(List[Int], Iterable[(Double,Int,Int)])]): RDD[(List[Int], Itera-
ble[(Double,Int,Int)])]={
    return rdd.filter(x => !(dominatedCellsB.value contains x._1))}
def addLocalDivisionPoints(rdd:RDD[Array[Double]])  = {
    rdd.map(x => x.zipWithIndex)
      .map(x => x.map(y => (y, divisionPointsB.value(y._2))))}
def getDominatedCells(list: List[List[Int]]): Array[List[Int]] ={
    var arraybuffer = ArrayBuffer[List[Int]]()
    list.foreach(x => { if (list.exists(l => isDominatedCell(x, l))) {arraybuffer += x }})
    return arraybuffer.toArray}
  def isDominatedCell(cell1: List[Int], cell2: List[Int]): Boolean ={
    var flag=true
    for(i<-0 to cell1.length-1)
    { if(cell1(i)<=cell2(i))
      {flag=false}}
    return flag
  }
}
```

simpleSkylineCalculation.scala

```
object skylineCalculation extends Serializable {

  def calculate(x: Iterator[Array[Double]]): Iterator[Array[Double]] = {
    var tempList = x.toList
    var i = 0
    var listLength = tempList.length
    while (i < listLength - 1) {
      var k = i + 1
      while (k < listLength) {
        if (dominationCondition.isDominated(tempList(i),tempList(k))) {
          tempList = tempList.take(k) ++ tempList.drop(k + 1)
          k = k - 1
          listLength = listLength - 1
        }
        else if (dominationCondition.isDominated(tempList(k),tempList(i))) {
          tempList = tempList.take(i) ++ tempList.drop(i + 1)
          listLength = listLength - 1
          i = i - 1
          k = listLength
        }
        k = k + 1
      }
      i = i + 1
    }
    return tempList.toIterator
  }
}
```

SFSSkylineCalculation.scala

```
object SFSSkylineCalculation extends Serializable {
  def calculate(x: Iterator[Array[Double]]): Iterator[Array[Double]] = {
    var arraybuffer = ArrayBuffer[Array[Double]]()
    val array = x.toArray
    arraybuffer += array(0)
    for (i<-1 to array.length - 1)
      {
        var j=0
        var breaked = false
        breakable
        {
          while (j < arraybuffer.length) {
            if (dominationCondition.isDominated(array(i), arraybuffer(j))) {
              arraybuffer.remove(j)
              j-=1
            }
            else if (dominationCondition.isDominated(arraybuffer(j), array(i))) {
              breaked = true
              break()
            }
            j += 1
          }
        }
            if(!breaked)
              arraybuffer+=array(i)
      }
    return arraybuffer.toIterator
}
  def addScoreAndCalculate(x: Iterator[Array[Double]]):Iterator[Array[Double]]={
    val y = addScoringFunction(x)
    val ysort = sortByScoringFunction(y)
    val result = calculate(ysort.map(x=>x._1))
    return result
  }
  }
```

```
def calculatePartition(previousSkylines: ArrayBuffer[Array[Double]], enteredPartition: Itera-
tor[Array[Double]]): Iterator[Array[Double]]= {
    var wasEmpty=false
    val array = enteredPartition.toArray
    if(previousSkylines.length==0){
        previousSkylines += array(0)
        wasEmpty=true
      }
    for (i <- 0 to array.length - 1) {
      var j = 0
      var breaked = false
      breakable {
        while (j < previousSkylines.length) {
          if (dominationCondition.isDominated(array(i), previousSkylines(j))) {
            previousSkylines.remove(j)
            j -= 1
          }
          else if (dominationCondition.isDominated(previousSkylines(j), array(i))) {
            breaked = true
            break()
          }
          if(wasEmpty & i==0)
            {
              breaked=true
              break()
            }
          j += 1
        }
      }
      if (!breaked) {
        previousSkylines += array(i)
      }
    }
    return previousSkylines.toIterator
  }
```

```
def sortByScoringFunction(iterator: Iterator[(Array[Double], Double)]):Iterator[(Array[Double],
Double)]=
  {
    var array=iterator.toArray
    array.sortBy(x => - x._2)
    return array.toIterator
  }

  def addScoringFunction(array:Iterator[Array[Double]]): Iterator[(Array[Double], Double)] ={
    array.map(x => (x, 0))
      .map(x => {
        var sum =0.0
        for (i<-0 to x._1.length - 1)
        {
          sum += math.log(x._1(i)+1)
        }
        (x._1,sum)
      })
  }
}
```

dominationCondition.scala

```
object dominationCondition extends Serializable {
  def isDominated(x: Array[Double], y:Array[Double]): Boolean = {
    return isSmaller(x,y) & isSmallerEqual(x,y)}
  def isSmaller(x: Array[Double], y:Array[Double]):Boolean = {
    val size = x.length
    var flag = false
    var i = 0
    for (i <- 0 to size - 1) {
      if (x(i) < y(i))
        flag = true}
    return flag}
  def isSmallerEqual(x: Array[Double], y:Array[Double]):Boolean = {
    val size = x.length
    var flag = true
    var i = 0
    for (i <- 0 to size - 1) {
      if (x(i) > y(i))
        flag = false}
    return flag
  }
}
```