Scalable processing of Dominance-Based Queries

|  |  |
| --- | --- |
| **Theodoros Konstantinidis - 64**  Computer Science Department  Aristotle University of Thessaloniki  Thessaloniki, Macedonia, Greece  theodorosk@csd.auth.gr | **Panagiotis Papaemmanouil - 56** Computer Science Department  Aristotle University of Thessaloniki  Thessaloniki, Macedonia, Greece  ppapaemm@csd.auth.gr |

# Introduction

This assignment required implementing an algorithm for the ***Skyline*** and ***Top-k dominant query*** problems. The algorithms must be scalable and efficient to handle large, multi-dimensional datasets. The ***Spark*** framework allows for scalability, whereas the ***Scala*** language is best for writing code, on top of the Spark framework, that is simple and easy to correct and maintain.

# Technology stack

## Spark

Solving the problem requires the use of a cluster-computing framework. Spark is an open-source distributed general-purpose cluster-computing framework, which provides an interface for programming entire clusters with implicit data parallelism and fault tolerance. It also supports a pseudo-distributed local mode, where no distributed storage is required and an executor is assigned per CPU core.

## Scala

The Spark framework, although written in Scala, supports a wide variety of programming languages, including but not limited to, Java, R, and Python. For this project, the Scala language was chosen to implement the algorithms, not only to fulfill the assignment’s requirements but also because Scala is a language of low complexity and is relatively safe to type in, while also retaining a perfect balance between productivity and performance.

# 

## KEYWORDS

Scala, Spark, skyline,dominance-based queries, distributed computing, high-performance computing.

# 1 Problem

Based on the assignment’s instructions the algorithm must perform 3 Tasks when given a set of d-dimensional points:

**Task1**. Given a set of d-dimensional points, return the set of points that are not dominated. This is also known as the ***skyline set***.

**Task2**. Given a set of d-dimensional points, return the k points with the highest dominance score. The dominance score of a point p is defined as the total number of points dominated by p.

**Task3**. Given a set of d-dimensional points, return the k points from the skyline with the highest dominance score.

## 1.1 Skyline Definition

The skyline term was first introduced by Borzsonyi [4] who issued the maximal vector computation problem in database applications.

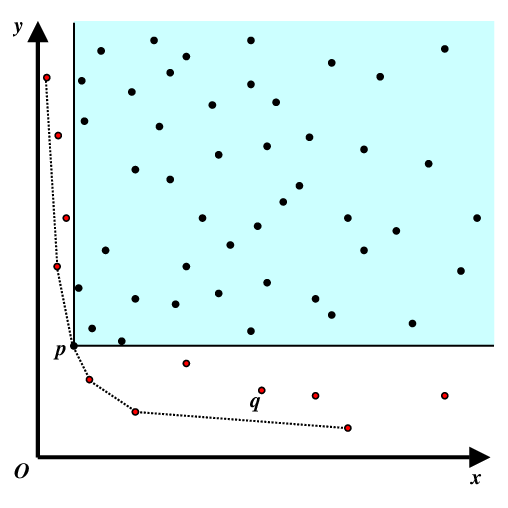
More specifically, given a set of tuples having an ordering relation on each dimension, Skyline is a subset of all the tuples that are not dominated by any other tuple of the original set. A tuple 𝑎 dominates another tuple 𝑏 (𝑎 ≺ 𝑏) when the values of each of 𝑎’s attributes are bigger (or smaller) than or equal to the corresponding values of 𝑏.

A real-world example where the skyline operator is applicable and useful is in the Real-Estate domain. The potential buyer tries to identify the best opportunity to rent or buy a property based on its characteristics. These characteristics might include the size of the property, the number of bedrooms, the number of bathrooms, the age of the building, etc, and finally the price. These characteristics are probably highly anticorrelated so the potential buyers trying to find a relatively cheap property with large size and as many bedrooms as possible can’t sort the properties based on any characteristic and get useful results. This is because if they sort the listings based on the ***price (ascending order, lower first)*** the results will probably have small size and not a lot of bedrooms. On the other hand, if they sort the listings based on the **s*ize (descending order - bigger first)***, the results will probably have a high price.

Here is when the skyline operator comes in hand. A skyline query can be set to maximize the size of the property and the number of bedrooms while at the same time minimize the price. In other words, ***the skyline operator will return all the listings which for them there is no other listing equal or better in all dimensions.*** Note: The “better” term depends on the type of variable, and can be mean lower or bigger.

In general, the Skyline problem is one of dominance. Point domination occurs when a point P is equal to a point Q in all dimensions and strictly better in at least one dimension.

In the following graph, the red points connected with the dashed line are the skyline set.



# 2 Related work

Even though the skyline problem has many practical applications, it hasn’t received much attention from the research community. Furthermore, even fewer publications can be found for the implementation of the skyline operator in a distributed environment.

In this section, we will describe the most common algorithms found in Literature, regarding the skyline operator implementation in a distributed environment and the pros and cons of every one.

In general, researchers aim to improve the performance of the skyline operator, focusing on 3 main directions:

1. Parallelize and speed-up calculations
2. Reduce the total dataset points using smart elimination methods
3. Reduce network traffic between machines.

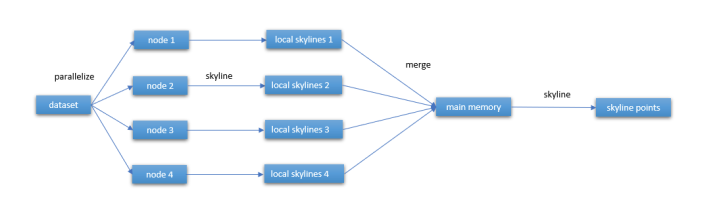
Based on these directions, there are 3 main algorithms proposed in the literature for this problem, each one of them exploiting a different aspect of the Spark Framework.

These algorithms are:

1. All Local Skyline (ALS)
2. Nested SQL Query using spark SQL library
3. Grid-partitioning algorithm

**All Local Skyline (ALS)**

This is a baseline approach. It 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 spark master which calculates the final skyline tuples. This approach called all local skylines (ALS) and 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.

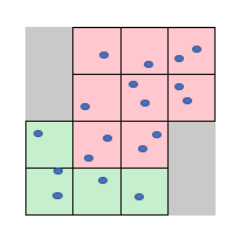


**Nested SQL Query using spark SQL**

This method exploits the spark SQL library and its optimization techniques. 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.

**Grid-partition algorithm**

This algorithm depends on smart elimination methods to reduce the total number of points in the dataset queried. It uses the Nearest-Neighbor methods to partition the dataset into regions and to exclude those regions that are evidently dominated by others.



**Methods to calculate skyline in a single-core machine**

The effectiveness of the algorithms presented above heavily relies on the local and centralized algorithms used for the skyline set identification.

Two main approaches for the skyline calculation were proposed in Literature.

The first, called ***simpleSkylineCalculation*** and 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 called ***SFSkylineComputation*** *(Sort first)*, and it is first implemented by Ilaria Bartolini [3]]. It first sorts the dataset in ascending order according to a monotone preference function. The first point is inserted into a candidate list and the components of the list are compared with the rest of the points. If a point dominates one or more points of the list, these points are deleted. If the point is not dominated by any point of the list, it is inserted in the list

**For our technical implementation, we use the ALS (All Local skyline algorithm) along with the SFSskylineCalculation method for the local skyline calculations.**

# 3 Implementing the algorithm

***Github repository:*** <https://github.com/papaemman/Big-data-Analytics-technologies-AUTh>

## 3.1 Main object / Initialization

The user interfaces with the ***all\_local\_skyline\_w\_topk.scala*** main object, its purpose being initializing and timing. The object accepts an array of arguments of undefined size as per its declaration. Valid arguments are the path of a valid dataset, the number K for getting the K points with the highest dominance score, both from the whole dataset and the skyline computed, and the number of cores used. Arguments ***must*** be entered in the order described above.

Before running any code in Scala over the Spark framework, a Spark Context is needed. The Spark Context is the main entry point for Spark functionality, as it represents the connection to a Spark cluster. to create one, a Spark Configuration (SparkConf) value is required.

In this configuration, setting the *master* to ‘local’ connects the Spark Context to a local cluster, enabling the aforementioned pseudo-distributed local mode. Within this specification, the number of cores is passed as an argument in the form of a string of characters complementing the ‘local’ string, augmenting it to ***“local[cores]”***. A name for the job is also required, which is set by setAppName().

The Spark Context is then created as an object with the Spark Configuration passed as an argument and can be used for actions within the pseudo-cluster by calling its functions or, since it is an actual object, it can be passed as an argument to another function, which is our case.

The Spark Context, filepath, and K variable are passed to the constructor of the ALTOPK class. Since the class contains no more than one function (the constructor itself) and no values have to be returned, the constructor is called without storing an object of the class.

## 3.2 Core function

The ***ALTOPK*** class is the actual code being executed when the job runs. It accepts the above arguments and contains commands for finding the dataset’s skyline and extracting a number of points with the highest dominance score, both from the whole dataset and the subset of points belonging to the skyline.

### 3.2.1 All Local Skyline

The code used for finding the skyline set is based on the All Local Skyline algorithm. It parallelizes the dataset in partitions, calculates a skyline for each one, then collects results and recalculates the skyline set, which is returned as its final result.

While it calculates each partition, it adds a score to each point, occurring from the sum of each point’s coordinates’ natural logarithms, which, mathematically, is a monotone function. According to this score, the points are sorted in descending order, then passed to another function to calculate which points are dominated and remove those from the partition.

A function is ultimately called which reads every partition and a static buffer, performing the same dominance comparisons to calculate the final skyline.

### 3.2.2 Top-K Points by Dominance Score

Unrelated to the skyline set, the extraction of the top K points by dominance score follows. For this task, the same domination comparison functions are used, however slightly modified. The domination score is no longer calculated according to a mathematical function. A natural number, the number of points dominated by a certain point P, is assigned to the latter as its score.

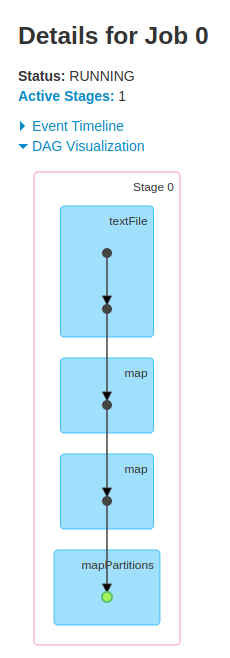
All points are then sorted in descending order according to their dominance score, but none are removed. Instead, the whole set is returned as a result. Selection of the top K points occurs within the core function and follows the previous actions, and since the points are already sorted, selecting the first K from the set returns the desired result.

### 3.2.3 Top-K Points by Dominance Score - Skyline

This time related to the skyline set, another extraction of top K points is performed. Using the results of the two previous tasks, a third set is created from the dataset sorted by dominance score. However, only the points belonging to the skyline are retained by using a filter function, thus returning the skyline sorted by dominance score as a result.

Extraction again occurs by selecting the first K points from the resulting set, since points are already sorted.

The **DAG (Directed Acyclic Graph)** formed by the spark in order to execute the skyline calculation (Task 1) and top-k dominance query (Task 2) is presented below.



# 4 Execution and results

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

As one can understand from the above definition, a distributed system consists of multiple machines. Unfortunately, this hardware setup (ie a cluster of computers) wasn’t available to us so we compromise and use the ***pseudo-distributed local mode*** offered by Spark, where no distributed storage is required and an executor is assigned per CPU core.

The Technical Details of our hardware setup can be found in the following table.

|  |  |
| --- | --- |
| **Table 1: *Machine Specifications*** | |
| Model name | Lenovo-V330-15IKB |
| OS | Ubuntu 18.04.5 LTS |
| Architecture | x86\_64 |
| Processor | Intel® Core™ i7-8550U CPU  @1.80GHz × 8 |
| CPU(s) | 8 |
| Memory | 7,1 Gib |
| GNOME | 3.28.2 |

The versions of Scala and Sparks were the following,

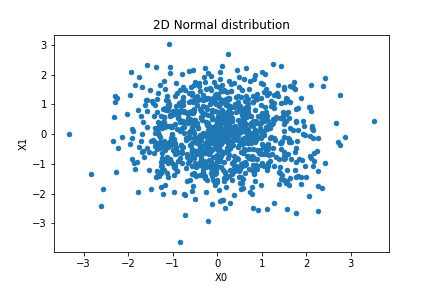
* ***Scala version***: 2.12.12
* ***Spark version***: 3.0.1

## 4.1 Datasets

In order for us to test the performance of our solution, we created synthetic datasets that were generated using the correlated, normal, uniform and anticorrelated distributions. We create these datasets using the **NumPy** Library in **Python** and more specifically the ***random*** module.

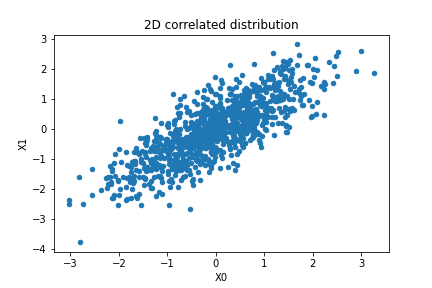
For the uniform distribution, we use the function ***numpy.random.uniform()*** independently for each dimension.

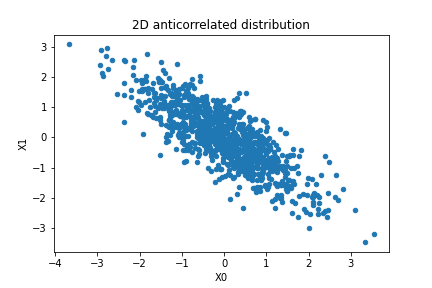
## 

For the Normal distribution, we use the ***numpy.random.multivariate\_normal()*** function, specifying the identity matrixThis is the rendered form of the equation. You can not edit this directly. Right click will give you the option to save the image, and in most browsers you can drag the image onto your desktop or another program. as the covariance matrix. 

For the correlated and anticorrelated distributions, we use the ***numpy.random.multivariate\_normal()*** function again, but specifying covariance matrix as

* This is the rendered form of the equation. You can not edit this directly. Right click will give you the option to save the image, and in most browsers you can drag the image onto your desktop or another program. for the correlated distribution
* This is the rendered form of the equation. You can not edit this directly. Right click will give you the option to save the image, and in most browsers you can drag the image onto your desktop or another program.for the anticorrelated distribution





Apart from the distribution variable, we construct the Experimental design to consists of 5 different variables, presented in the table below.

|  |  |
| --- | --- |
| **Table 2: Variables (*Degrees of Freedom) in Experimental Design*** | |
| Total points | 10.000, 100.000, 1.000.000 |
| Dimensions | 2, 4, 10, 50 |
| Distribution | Uniform, Normal, Correlated, Anticorrelated |
| k (top-k) | 1, 10, 50, 100 |
| cores | 1, 2, 4, 8 |

**Note:** Our initial planning was to run 3 x 4 x 4 x 4 x 4 = ***768 experiments*** and we had already implemented the infrastructure and datasets to do it, unfortunately, due to time constraints, we managed to run **162 experiments.**

* **Total Experiments: 162**

In the following table, the total number of experiment runs per category presented for every variable.

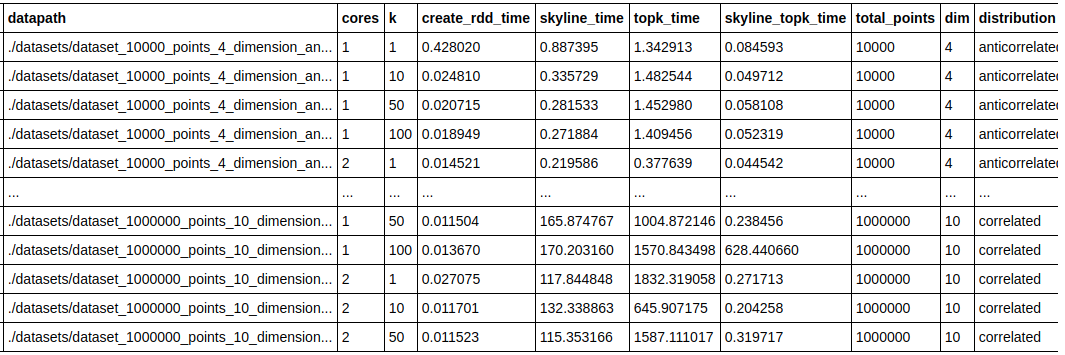
***Total Experiments per variable***

|  |  |
| --- | --- |
| total points | 1. ***10.000 points***: 108 2. ***100.000 points***: 36 3. ***1.000.000 points***: 18 |
| Dimensions | 1. ***2****: 2* 2. ***4***: 64 3. ***10***: 80 4. ***50***: 16 |
| Distribution | 1. ***Uniform***: 42 2. ***Normal***: 28 3. ***Correlated***: 56 4. ***Anticorrelated***: 36 |
| k (top-k) | 1. ***1***: 65 2. ***10***: 32 3. ***50***: 32 4. ***100***: 32 |
| cores | 1. ***1 core***: 42 2. ***2 cores***: 40 3. ***4 cores***: 40 4. ***8 cores***: 40 |

For every experiment, we measure the time for each one of the 3 Tasks separately. So for every experiment, the reported times were:

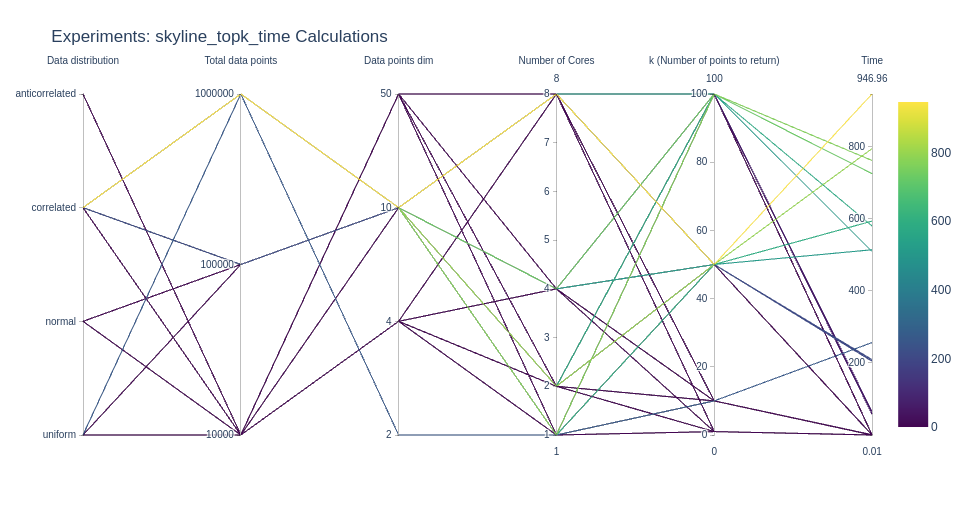
1. ***skyline\_time (Task 1)***
2. ***topk\_time (Task 2)***
3. ***skyline\_topk\_time (Task 3)***

***Experiments results (sample table)***



Based on our experimentation process we derived insights related to how every variable (degree of freedom) in our experimental design affects the performance of each individual task.

To do this we leverage 2 kinds of **Visualization techniques**. The first one is the Parallel Coordinates plot. This plot demonstrates every experiment as a zig-zag line and every variable as a parallel vertical line. In addition, every line is colored based on the total time of experiments. As one can understand this plot was a useful tool in order for us to identify how each variable affects the running time.

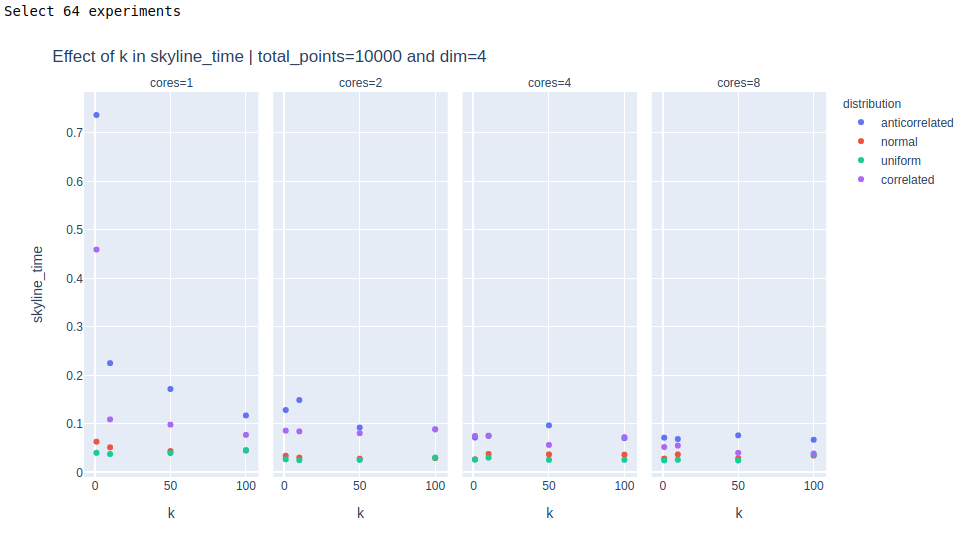


The second plot we use represents every experiment as a point in a 2-dimensional plane. We use the number of cores as the discriminative variable for the 4 different panes, the distribution variable as the color of the dots, and the time as the variable in the vertical axis.

To evaluate the **effects** of the remaining 3 variables, namely:

* k
* total\_points
* dimensions (dim)

we create 3 different plots at which, every time we fix the 2 of these variables and we use the 3rd as the variable in the x-axis.

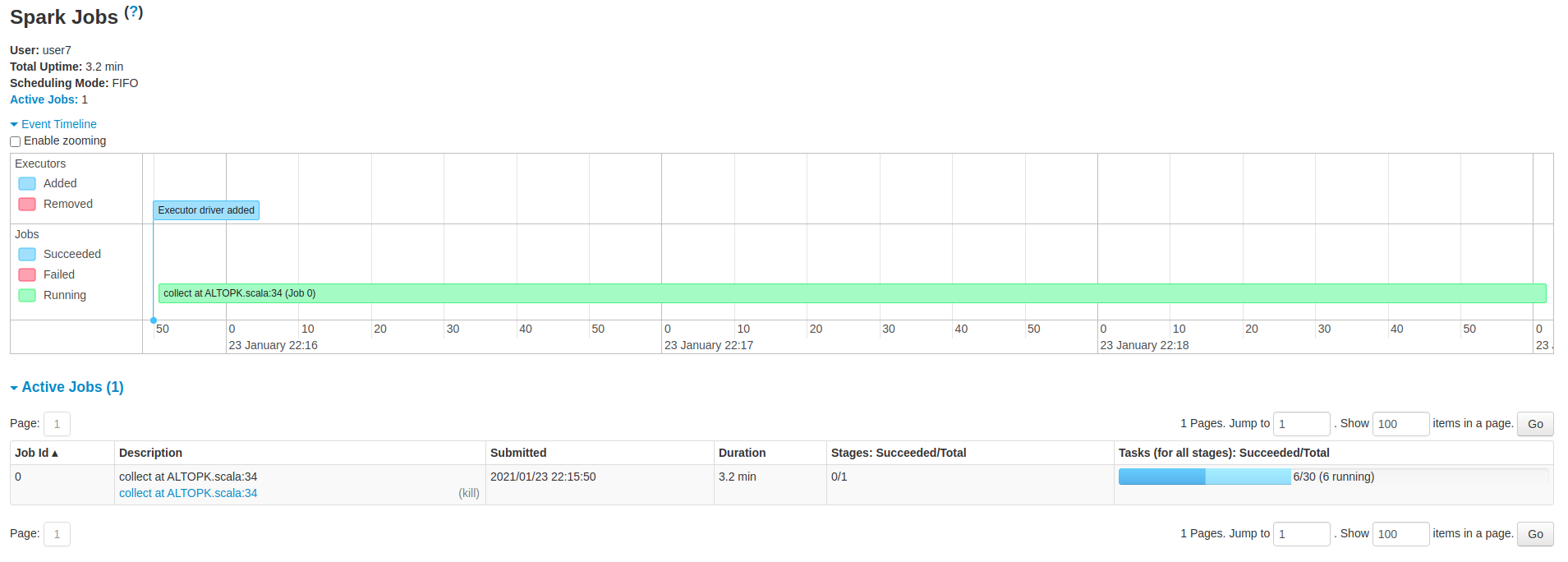


Based on the process described above we found out that:

1. The ***topk*** algorithm (Task 2) required significantly more time than the skyline algorithm (Task 1). This is because the skyline implementation used a sorting step before calculating local skylines while the topk algorithm compares each point with each other, ie. it has O(n^2) complexity.
2. The number of dimensions (dim) affects exponentially the running time because it increased the time to test the domination condition between 2 points.
3. As the number of cores increasing the running time decreasing if the dataset is large enough. We encounter the greatest difference running the same experiment using 1 core and 8 cores, for a dataset with 1.000.000 points and 50 dimensions.
4. The parameter k seems to not affects the running time.
5. Depending on the task some distributions were evidently faster than others. The anticorrelated distribution was slower for the skyline calculation task.

**Additional Note: Spark UI**

During the experimentation process, we heavily use the ***Spark UI*** offered by the spark, in order to monitor the state of the experiments. A screenshot example of this can be seen below:



**Github repository**

The code described above, as long as the experiments results and their analysis can be found in our GitHub repository at the following link:

<https://github.com/papaemman/Big-data-Analytics-technologies-AUTh>

# 6 References

The following Literature can be found in our GitHub repository under the path ***/docs/bibliography***.

[1] Papanikolaou Ioanna, 2018. Distributed algorithms for skyline Computations. *International Hellenic University.*

[2] Guidan Chen, Yongheng Wang. 2018. Top-k Dominating Query Processing over distributed data streams. *Global Journal of Engineering Science and Research Management.*

[3] Bartolini, I., Ciaccia, P., and Patella, M. 2008. Efficient sort-based skyline evaluation. ACM

Trans. Datab. Syst. 33, 4, Article 31 (November 2008), 49 pages. DOI = 10.1145/1412331.1412343.

<http://doi.acm.org/10.1145/1412331.1412343>

[4] Stephan Borzsonyi, Donald Kossmann, Konrad Stocker. The skyline operator.

[5] Eleftherios Tiakas, Apostolos N. Papadopoulos, Yannis Manolopoulos. Top-k Dominating Queries: a Survey

[6] Man Lung Yiu, Nikos Mamoulis. Multi-dimensional top-k Dominant Queries.

[7] Katja Hose, Akrivi Vlachou. A Survey of Skyline Processing in Highly Distributed

Environments.

[8] Md. Anisuzzaman Siddique, Hao Tian, Mahboob Qaosar, Yasuhiko Morimoto (2019). MapReduce Algorithm for Variants of Skyline Queries: Skyband and Dominating Queries

DOI:<https://www.mdpi.com/1999-4893/12/8/166>

[9] The skyline problem. [The Skyline Problem | Divide And Conquer | learnbay.io](https://www.learnbay.io/the-skyline-problem/)