Skyline and Top-k Most Dominant Skyline Points

Dimitris Tourgaidis
Data and Web Science,
School of Informatics
Aristotle University of
Thessaloniki

Thessaloniki Macedonia
Greece
tourgaidi@csd.auth.gr

Introduction

For the purposes of the class, I fulfilled the project that was assigned to us and present it through this report. My objective was to implement the required algorithms as efficient as possible, furthermore, to execute the required variety of scenarios, based on the project's instructions.

The structure of the report is the follow. At section 1, I explain how I generated all the datasets, as well the different scenarios to which the generated datasets are related. Then at section 2, for each task I present the algorithms and their implementation in Spark, as well their execution times for all the different scenarios. Last, at section 3, I present some selective results of the algorithms executions.

1 Dataset Generation

In order to generate the datasets that I used to execute the different required scenarios, I used python's library Numpy. The datasets are containing n-dimensional points and below we can see the first 3 lines of a file, that I produced, that contains three 2-dimensional points, where each line shows the coordination of a point. For space purposes, the below numbers are presented rounded, which I don't do in my datasets.

2.702 2.231

2.650 2.219

2.582 2.079

Using Numpy, I generated data, for 4 different distributions; correlated, anticorrelated, uniform and normal. At first, for each distribution, I created a number of different datasets for 3 different point-dimensionality. The datasets were related to 2-dimensional, 4-dimensional and 8-dimensional points. Then, for each distinct point dimensionality, I created datasets that contained 10, 100, 1000, 10000, 100000 and 1000000 points. Consequently, for each distribution, I executed each algorithm that I implemented, for 18 different scenarios, based on the different combinations related to the point-dimensionality and the number of points in a dataset. Furthermore, due to the fact that I executed the aforementioned scenarios for 2 different number of cores cases, one with 1 core and then with 4 cores, at the end, I executed 36 scenarios for each distribution.

2 Tasks Implementation

In this section I present and analyze the algorithms of the tasks that I solved. Furthermore, I present how I implemented the algorithms in Spark and I conclude with presenting the execution times for each task for the different scenarios I executed.

2.1 Implementation of Task 1

To solve task 1, I use the All Local Skyline (ALS) approach. At first, I calculate the local skylines points in each partition, then collect them together in the driver and then calculate the global skylines points. Figure 1, shows how I implemented ALS in Spark, through an example where the RDD is partitioned into 3 partitions.

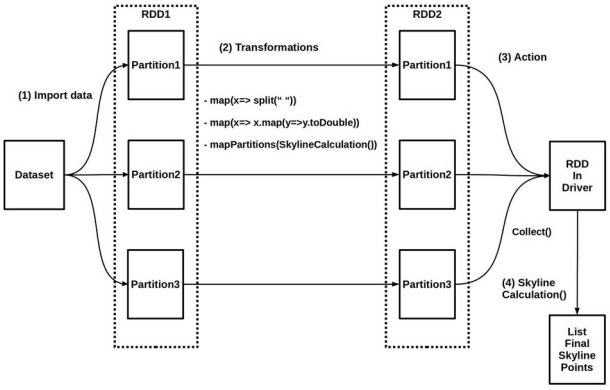

Figure 1: **Spark implementation of task 1 algorithm**

Analyzing Figure 1 from left to right, we can see that the implementation is divided by 4 stages. At stage (1) the dataset is imported and then distributed into the 3 partitions. Then, at stage (2), all the transformations are applied, where at first, map(x=> x.split(" ")) transforms each line of the dataset from a string to a List that contains the coordination of a point, with length n, where n is the number of dimensions of the points. Then, map(x=>x.map(y=>y.toDouble()) converts all the elements of the Lists from string to double and then through mapPartitions(SkylineCalculation()), a SkylineCalculation() function, implemented by me, is executed in each partition that calculates the local skyline points for each partition. Afterwards, at stage (3), an action is called which triggers all the aforementioned transformations, resulting to RDD2, whose partitions are containing the local skylines points for the corresponding partitions of RDD1. Through collect(), all the partition elements of RDD2, are collect at the main driver. At last, at stage (4), the SkylineCalculation() is called for the RDD in the main driver, consequently the global skyline points are finally calculated.

2.1.1 Algorithm optimization. As mention before, in order to calculate the local and global skyline points, I implemented a SkylineCalculation() function. In this function, I implemented 2 optimizations, in order to avoid a brute force comparison among all the points, to see if a point dominates an other one. Concerning the first optimization, firstly, I put all the points of the partition in a temporary list and afterwards I compare sequentially each point with all the other points. In case where a point dominates an other one, the dominated point gets extracted from the temporary list. Consequently, points that we know that can't belong in the skyline, are never compared to other points again, so unnecessary comparisons are avoided. Related to the second optimization, intuitively, we can assume that most of the points that belong to the skyline, are more likely nearer to the axis origin compared to the ones that not belong in the skyline. So, I do not put the points in the temporary list like the order they were placed in the partition, but firstly I find each point's distance from the axis origin and then put them in ascending order in the temporary list. In this way, it is much more likely that the first points in the temporary list, that are going to be compared with all the other points, to belong in the skyline or be the most dominant points, so many points are going to be extracted over the initial iterations from the temporary list, fact that decrease significantly the comparisons, consequently the execution of the algorithm. This axis origin optimization has a necessary condition. All the values of the point-dimensions must be positive or negative. In case where this condition is not met, a prior preprocessing could solve the problem.

2.1.2 Algorithm execution times. After the implementation of the algorithm, I ran the scenarios discussed in section 1. I executed the algorithm localy and the hardware of the machine in which I executed the different scenarios is present by the below bullets.

- i5-8265U, 1.60GHz , 4 Cores
- Ram 8GB

Figure 2 shows the execution times of the task 1 algorithm, with 1 core, for the different scenarios, and Figure 3 shows the corresponding results with 4 cores. The columns of the tables are related to the dimensionality of the points, the rows to the number of points in the dataset and the results of the execution times are presented in seconds.

	2	4	8
10	1.094	1.043	1.339
100	1.025	1.057	1.288
1000	1.167	1.212	1.267
10000	1.374	1.643	1.728
100000	1.705	3.238	4.477
1000000	4.693	23.592	-

Table1. Execution times for correlated distribution, with 1 core.

	2	4	8
10	1.179	1.282	1.144
100	1.285	1.269	1.315
1000	1.404	1.306	1.367
10000	1.593	1.900	2.543
100000	2.316	10.366	22.893
1000000	32.475	1075.755	-

Table2. Execution times for anticorrelated distribution, with 1 core.

	2	4	8
10	1.110	1.056	1.087
100	1.198	1.051	1.290
1000	1.351	1.189	1.632
10000	1.511	1.540	73.725
100000	2.946	10.273	3747.401
1000000	15.226	468.859	-

Table3. Execution times for uniform distribution, with 1 core.

	2	4	8
10	1.193	1.183	1.205
100	1.170	1.209	1.356
1000	1.239	1.350	2.041
10000	1.325	1.974	76.654
100000	2.272	16.882	2571.201
1000000	25.342	440.099	-

Table4. Execution times for normal distribution, with 1 core.

Figure 2: **Execution times of task 1 implementation, with 1 core.**

	2	4	8
10	1.203	1.137	1.271
100	1.260	1.199	1.061
1000	1.282	1.280	1.220
10000	1.380	1.523	1.735
100000	1.922	2.608	3.384
1000000	3.502	14.172	-

Table1. Execution times for correlated distribution, with 4 cores.

	2	4	8
10	1.154	1.253	1.124
100	1.140	1.067	1.134
1000	1.187	1.683	1.285
10000	1.360	1.919	1.805
100000	2.300	6.046	16.955
1000000	17.982	667.185	-

Table2. Execution times for anticorrelated distribution, with 4 cores.

	2	4	8
10	1.219	1.227	1.188
100	1.218	1.277	1.241
1000	1.253	1.277	1.567
10000	1.425	1.449	44.796
100000	2.248	10.353	2378.606
1000000	8.711	248.894	-

Table3. Execution times for uniform distribution, with 4 cores.

	2	4	8
10	1.274	1.252	1.304
100	1.246	1.267	1.276
1000	1.197	1.436	1.930
10000	1.427	1.755	45.020
100000	2.480	9.009	1691.980
1000000	16.023	179.037	-

Table4. Execution times for normal distribution, with 4 cores.

Figure 3: **Execution times of task 1 implementation, with 4 cores.**

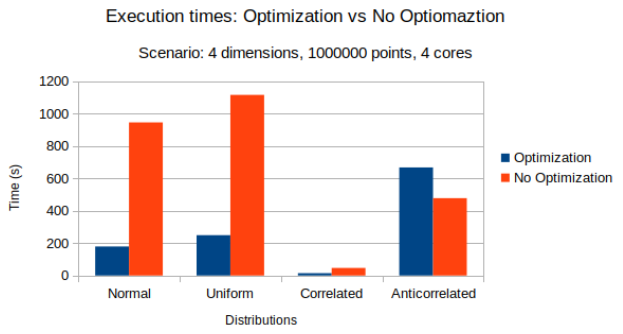

Figure 4: **Execution times with and without the axis origin optimization.**

All the above executions times, are results of the algorithm using the axis origin optimization. In order to measure the level of optimization of the axis origin optimization that I implement, I executed 1 selective scenario for each distribution without the optimization and compared the results, with the corresponding execution times, where the optimization was used. The scenario for which I made the comparison is for 4-dimensional points in a dataset that contains 1000000 points. Figure 4 shows the results of the comparison.

As Figure 4 shows, the axis origin optimization provided remarkable optimizations to the uniform and normal distributions. For the case of the correlated distribution, the results are slightly better, but on the other hand, the axis origin optimization results to worst execution times for the anticorrelated distribution in scenarios related to datasets with number of points more than 100000.

2.2 **Implementation of Task 2**

To solve task 2, I calculate the global skyline points, then compare them with all the other points, in order to calculate their dominance score and finally extract the k most dominate. Figure 5 and Figure 6, show how I implemented this algorithmic approach in spark.

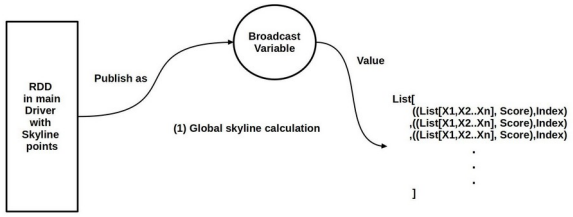

Figure 5: **Stage (1), of Spark implementation of task 2 algorithm**

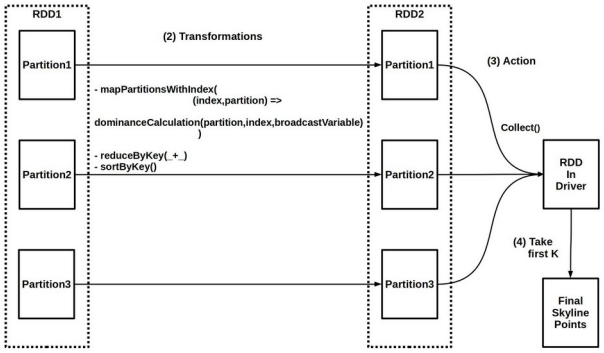

Figure 6: **Stage (2) (3) (4), of Spark implementation of task 3 algorithm**

Analyzing Figure 5 and Figure 6, we can see that the implementation is divided by 4 stages. At stage (1), the skyline points are calculated using the SkylineCalculation() function of task 1, but with a slightly different implementation. Instead of mapPartitions, I used mapPartitionsWithIndex, so the SkylineCalculation() function takes as arguments the partition and its index. The result of the SkylineCalculation() is not just the local skyline points, as it was in task 1, but also their dominance score in the partition, as well the index of the partition they belong. Below is presented the key-value pair produced for a skyline point, after the mapPartitionsWithIndex transformation, solely the execution of the skylineCalculation() function in each partition.

((List[X0,X1….Xn], Score), Index)

The *List[X0,X1….Xn]* represents the coordination of the point, *Score* represents its dominance score in the partition and *Index* represents the index of the partition it belongs. The produced RDD from stage (1) is then distributed as a broadcast variable to all the workers and has the below structure.

List(

((List[X0,X1….Xn], Score), Index)

((List[X0,X1….Xn], Score), Index)

((List[X0,X1….Xn], Score), Index)

.

.

)

Then, at stage (2), transformations are applied to the initial RDD, which contains all the points of the imported dataset. At first, through mapPartitionsWithIndex, a dominanceCalculation() function, implemented by me, is executed in each partition. The results of dominanceCalculation(), being executed in a partition is that the elements of the broadcast variable, except from those that their partition index is equal to the index of the partition, are compared with all the points of the partition. I avoid to compare skyline points again with points that they have already been compared, through the *Index* value of the key-value pair they are encoded. All the comparisons are resulting to a key-value pair presented below.

(Index,0)

or

(index,1)

The result of a comparison between a skyline point and another point is *(Index,0),* if the skyline point doesn’t dominates the other point, where *Index* is the index value of the skyline point in the broadcast variable, which is a list. On the other hand, the result is *(Index,1),* if the skyline point dominates the other point. Consequently, the execution of the dominanceCalculation() function in a partition results to a RDD that contains the produced key-value pairs, analyzed before, plus key-value pairs for the skyline points that are stored in the partition and they dominance score for the partition was already calculated at stage (1). The key-value pairs for those cases are presented below.

(Index,Score)

The *Index* represents the index of the skyline points in the broadcast variable and *Score* their dominance score in the partition. Figure 7 shows the produced RDD after the execution of dominanceCalculation() function in a partition.

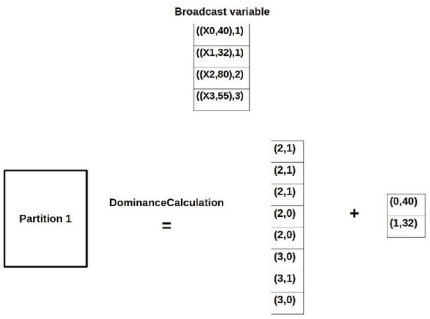

Figure 7: **Results of dominanceCalculation() function in a partition.**

In Figure 7, the results of the first list, are the dominance comparison results of the third and fourth skyline point in the broadcast variable, which have different partition index compared to the partition to which the dominanceCalculation() function is executed. The results of the second list, are the dominance score of the first and second elements of the broadcast variable, which have the same partition index as the partition i which the dominanceCalculation() function is executed.

After the execution of dominanceCalculation() function in each partition, a reduceByKey() transformation is applied, which aggregates the values of the key-value pairs with the same key, solely calculated the dominance score for each skyline point. Next, a sortByKey() transformation is applied, due to the fact that we want to find the K most dominate points. At stage (3), the action function collect() is applied, triggering all the aforementioned transformations and resulting to collect all the skyline points and they dominance scores in the main driver. At last, at stage (4) the first k elements of the RDD in the main are been taken, solely the k most dominate points.

2.2.1 Algorithm execution times. After the implementation of the algorithm, I ran the scenarios discussed in section 1. I executed the algorithm locally and the hardware of the machine in which I executed the different scenarios is present by the below bullets.

- i5-8265U, 1.60GHz , 4 Cores
- Ram 8GB

Figure 8 shows the execution times of the task 3 algorithm, with 1 core, for the different scenarios, and Figure 9 the corresponding results with 4 cores. In all the executions, I set the algorithm to return the 10 most dominance points. Furthermore, in the case of task 3, I didn't executed scenarios regarding 1000000 points. The reason was the time those scenarios needed to be executed, resulting to an extended overheat to my laptop, solely damage, that that I couldn't afford. The columns of the tables in the Figure 8 and Figure 9, are related to the dimensionality of the points, the rows to the number of points in the dataset and the results of the execution times are presented in seconds.

	2	4	8
10	1.850	1.825	1.905
100	1.862	2.037	2.039
1000	2.040	2.173	2.447
10000	5.506	6.487	21.089
100000	103.509	517.224	3114.865

Table1. Execution times for correlated distribution, with 1 cores.

	2	4	8
10	1.937	2.063	1.935
100	1.885	2.053	2.021
1000	2.002	2.424	2.532
10000	4.281	10.024	41.024
100000	259.572	1801.876	-

Table2. Execution times for anticorrelated distribution, with 1 cores.

	2	4	8
10	1.860	1.866	1.954
100	1.773	1.973	2.109
1000	1.987	2.582	3.035
10000	10.919	16.907	266.026
100000	217.785	2362.744	-

Table3. Execution times for uniform distribution, with 1 cores.

	2	4	8
10	1.891	1.921	1.968
100	1.914	1.889	1.961
1000	1.990	2.403	3.199
10000	6.740	23.377	257.593
100000	174.489	3693.778	-

Table2. Execution times for normal distribution, with 1 cores.

Figure 8: **Execution times of task 2 implementation, with 1 core.**

	2	4	8
10	2.116	2.139	2.214
100	2.158	2.239	2.220
1000	2.254	2.418	2.552
10000	3.007	4.835	11.533
100000	55.708	307.679	4753.911

Table1. Execution times for correlated distribution, with 4 cores.

	2	4	8
10	2.208	2.214	2.081
100	2.291	2.223	2.311
1000	2.358	2.491	2.792
10000	4.474	10.809	27.341
100000	337.310	785.664	-

Table2. Execution times for anticorrelated distribution, with 4 cores.

	2	4	8
10	2.049	2.100	2.113
100	2.144	2.155	2.355
1000	2.259	2.357	3.327
10000	3.395	17.129	175.546
100000	184.383	1512.548	-

Table3. Execution times for uniform distribution, with 4 cores.

	2	4	8
10	2.142	2.306	2.180
100	2.236	2.330	2.342
1000	2.313	2.619	3.583
10000	3.885	15.780	166.445
100000	115.316	1871.170	-

Table2. Execution times for normal distribution, with 4 cores.

Figure 9: **Execution times of task 2 implementation, with 4 cores.**

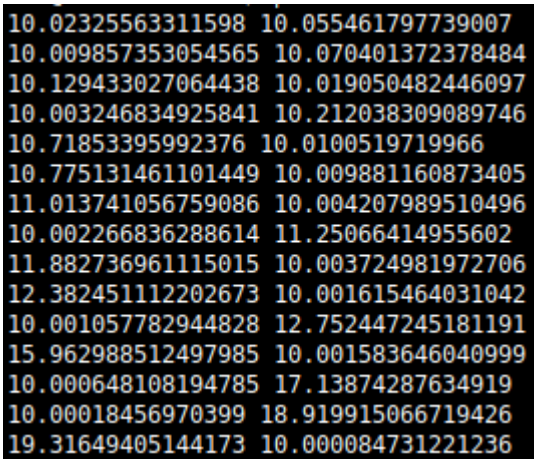

Figure 10: **Skyline points in a dataset with uniform distribution that contains 100000 2-dimensional points.**

3 Algorithm Execution Results

In this section I present the results of some selective executions of the 2 tasks. Due to the fact, that I executed many different scenarios, the presentation of all the execution results is impossible, for reasons related to the size of the report.

Regarding the results of task 1, Figure 10 shows the skyline points for the scenario related to a uniform distribution with 100000 2-dimensional points and Figure 11 shows the results for a correlated distribution with 1000000 4-dimensional points. In each Figure, each line represents a distinct point.

Figure 11: **Skyline points in a dataset with correlated distribution that contains 1000000 4-dimensional points.**

As for task 3, Figure 12 presents the results for a dataset with normal distribution with 100000 4-dimensional points and Figure 13 presents the results for a dataset with correlated distribution with 100000 4-dimensional points.

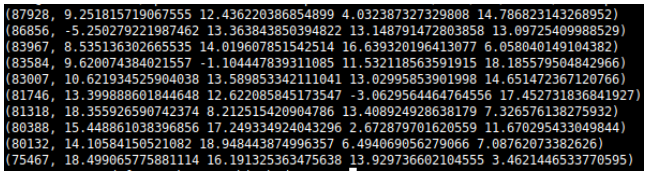

Figure 12: **Top-10 most dominant Skyline points in a dataset with normal distribution that contains 100000 4-dimensional points.**

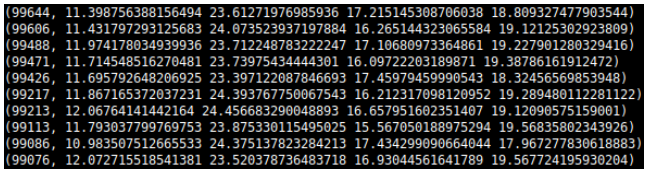

Figure 13: **Top-10 most dominant Skyline points in a dataset with correlated distribution that contains 100000 4-dimensional points.**

In each of Figure 12 and Figure 13, the results are presented in key-value pairs. The value in a key-value pair shows the point-coordination and the key the number of points it dominates.