Scalable Processing of Dominance-Based Queries

Technologies for Big Data Analytics 2020-2021

Konstantinos Giantsios
M.Sc. Data and Web Science
Aristotle University
54124 Thessaloniki, Greece
giantsik@csd.auth.gr

Vasiliki Lavrentiadou
M.Sc. Data and Web Science
Aristotle University
54124 Thessaloniki, Greece
vlavrent@csd.auth.gr

ABSTRACT

The following report focuses on the exploration and discovery of Skyline and top-k dominating points, utilizing the Big Data unified processing engine Spark. The data point creation , as well as an analytical explanation of the methods, employed for each task and their results, are thoroughly presented.

Introduction

The skyline and top-k dominating queries have been the center of a plethora of scientific researches. Computer and Data scientists have occupied themselves with the task of developing efficient algorithms, capable of accelerating the performance, both in terms of speed and complexity. A wide range of solutions, tackling the problem of skyline and top-k dominant queries, have been proposed so far.

As part of the Big Data Analytics project, the tasks of skyline objects and top-k dominating queries have been resolved. Moreover, the spark distributed computing system has been utilized for the completion of all tasks. Beside the creation of two sets of data, all tasks were developed with Scala.

In the following chapters, the resolution process for each task is introduced. The data creation as well as the algorithms followed will be presented. Additionally, conclusions regarding the results, execution time and supplementary diagrams are listed.

1.System Resources

For the execution of each specific task, a variety of CPU cores and memory were tested . After plenty of tests it was observed that, although the use of fewer CPU cores was unproblematic for lower dimensions, and most specifically for the second dimension datasets, it caused various difficulties when the code ran larger datasets of multiple dimensions. Therefore the code for each task was executed on the maximum CPU and memory capacity.

For each execution of the code the runtime was monitored and returned. For each dimension the correlation between time and volume of data is observed, via line plots.

1.1 Skyline and Top-k Skyline points

The Skyline code and Top-k Skyline points code were executed on a System with an AMD Ryzen 5 Processor of 6 cores, 12 logical processors and memory of 8GB. Although for each distribution datasets of 10.000, 50.000, 100.000, 500.000 ,1.000.000 and 10.000.000 were created. For the 10.000.000 data points the only runtime results are exported only from the 2-dimensional distributions, as it took too long to run the code for 10.000.000 data points as the dimensions increased. However 10.000.000 datasets were executed on 3 and 4 dimensional data and conclusions will be discussed in the end. Additionally, the skyline was also executed on 2 CPU cores only for the 2 and 3 dimensional data . The runtime was calculated in nanoseconds and converted to minutes.

1.2 Top-k dominating points

The experiments for task 2 (Top-k dominating points) were executed on a Manjaro 20.2.1 System with an AMD Ryzen 5 Processor of 6 cores, 12 logical processors and memory of 16GB.

2.Data

The datasets for computing the skyline and top-k dominating queries, were created according to four distributions, Correlated, Uniform, Normal and Anti-Correlated. Correlated and Uniform datasets were developed with scala whereas Normal and Anti-Correlated were developed with python. Correlated and Anti Correlated datasets were generated using Cholesky transformation. For each distribution up to 4-dimensional data were generated. Datasets regarding 10.000,50.000, 100.000, 500.000, 1.000.000 and 10.000.000 points were created for each distribution. Each dataset consists of data points that are in the form; (0,1,id) for 2-dimensional data, (0,1,2,id) for 3-dimensional

data and (0,1,2,3,id) for 4-dimensional data. The data values are in the range [0,100]. The datasets are saved in CSV files.

3.Code

As it has already been mentioned, the solving of skyline and top-k queries problems, were developed in Scala and executed in Spark.

3.1 Skyline

The data points, which are not dominated (from any other data point), define skyline. A data point is not dominated if it's coordinates are not greater than the coordinates of another data point. If two coordinates (or more) are equal ,then the other ones determine dominance. For the computation of the skyline, the Sort filter algorithm as well as the R-tree were examined. However, the computation of the skyline is based on the Sort Filter algorithm.Skylines are firstly computed locally, in each partition and are later returned on the driver where the global skyline is computed.

Sort Filter Algorithm

1. Dataset T, dimensions d, Skyline S
2. Sort all records by the sum of all dimension attributes in ascending.
3. Initialize S and move first record of T into S.
4. For each point t in T compare t with all records
    a. If t is dominated by a point in S, continue iteration with the next record.
    b. Else add t to S.
5. Return S

Sort filter algorithm analyzed

Firstly the data points are imported into an rdd. Each partition performs a computation that searches for the local skyline. In each partition the data points are transformed into double,(with a map transformation applied to the rdd) and each point is converted into a Tuple of k where k is the number of dimensions plus the id, plus the sum of the values of all dimensions (another map transformation is applied on the rdd for the Tuple creation). Having created the Tuples, the data are sorted in ascending order, according to the sum (last tuple). Afterwards, a mapPartitions transformation is applied on the two previous map transformations, in order to compute the local skylines.

2-dimensional data

In 2-dimensional data a Tuple of 4 values is created, in each partition, where Tuple4 equals (0,1,id,sum(v(0),v(1))). All partitions are sorted, according to the 4th Tuple, which is the sum of the two dimensions of each row of data. Subsequently, a list that represents the local skyline is created and the first row of Tuples is assigned to it. As long as the rdd is not empty, each point is compared with the points in the Skyline. For example a point a(0,1,id,sum(v(0),v(1))) belongs to the local skyline if:

a(0) <= S(0) and a(1)<=S(1)

where S represents all the Skyline points.

The point a(0,1,id,sum(v(0),v(1))) is not dominated by any of the local Skyline points and therefore it belongs to the local Skyline list. If a point is dominated by any of the Skyline points then the iteration and the comparisons between the point and the Skyline points stop. The next point is examined until there are no more points left.

Tuples are compared to the Tuples that belong in the skyline, until they find values that dominate them or not. If a point is not dominated by any skyline point, then it belongs to the local skyline and it is added to the local skyline list.

3-dimensional data

In 3-dimensional data, a Tuple of 5 is created where Tuple5 is (0,1,2,id,(v(0)+v(1)+v(2))). The data are sorted according to the 5th Tuple. A list containing the first point of the sorted rdd is created in all partitions. Each partition computes a local skyline by comparing each dimension value to the skyline points.

For example a point b(0,1,2,id,sum(v(0),v(1),v(2))) is a Skyline point if:

b(0)<=S(0) and b(1)<=S(1) and b(2)<=S(2)

where S represents all the skyline points.

Point b is not dominated by any of the Skyline points and therefore is a Skyline point, so it is added on the local Skyline list.

As it has been mentioned, points are compared until a point is dominated, or the skyline points, or rdd points, run out . All the local skylines are stored in lists in each partition.

4-dimensional data

In 4-dimensions a Tuple of 6 is created in each partition. Tuple6 has the form (0,1,2,3,id,(v(0)+v(1)+v(2)+v(3))). The data points are sorted according to the 6th tuple (sum of values of dimensions). After the data are sorted according to the sum of all dimensions, the first point is added to the skyline list. In each partition local skylines are computed by comparing the first point of the dataset with the first point of the skyline.An rdd point c(0,1,2,3,id,sum(v(0),v(1),v(2),v(3))) belongs to the local Skyline if:

c(0)<=S(0) and c(1)<=S(1) and c(2)<=S(2) and c(3)<=S(3)

where S represents all the skyline points.

Tuple c dimension values are smaller than all the dimension values of each skyline point, so it is added to the local Skyline list.

After the process of computing the local skylines in each partition is finished, the results are collected in the driver and an additional computation of the global skyline is performed. The Sort filter algorithm is once more applied on the list of collected Skylines, and the global skyline is formed. The global skyline is stored in a

list. The list is serialized and the data are transformed into dataframes in order to be saved in CSV files.

3.2 Top-k Dominating points

In this task, we have to find the top k dominating points, which means the points that dominate the more points. As we said in 2.1 a point dominates another point if the first's point coordinates are greater than the second's in all dimensions. For this task we adopted an algorithm and a practice using a grid over the given data points [3]. Before we present the algorithm first we should define some axioms.

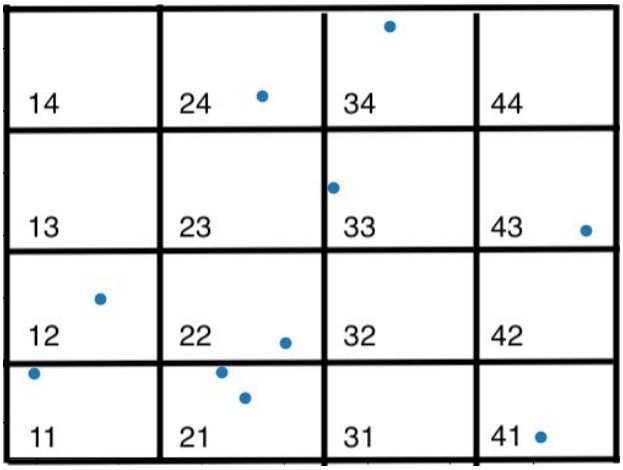

Figure 1: A 4x4 grid over a dataset

Axiom 1. Given a grid cell g, its upper bound score $r^u(g)$ is the total point count of cells it partially or fully dominates. A partially dominated point on the grid is a point that its cell shares the same value with the cell of a partially dominant point p for one or more dimensions and has greater or equal coordinates for the others dimensions. Accordingly, a fully dominated point on the grid is a point that its cell has lesser values on its coordinates of all dimensions than the cell of a fully dominant point. For example the grid cell 22 in figure 1 has one partially dominated point and three fully dominated points so its upper bound score is four.

Axiom 2.

Given a grid cell g, its lower bound score $r^l(g)$ is the total point count of cells it fully dominates. For instance the grid cell 22 in figure 1 has a lower bound score equal to three.

Axiom 3.

Given a grid cell g, g.f is its total point count of cells fully dominating g. For instance for cell 33 its g.f is equal to 5.

Axiom 4.

Cells with zero count and cells with g.f < k are not accounted as cells that we must check (eg. 33).

Axiom 5.

The prune score of a grid ps is defined as the lowest upper score of remaining grid cells. Those cells that have a lower bound score than the prune score can be pruned.

Filter Algorithm:

1. Dataset T, Grid G
2. Count number of points for every cell in G
3. Produce candidates grid cells based on axiom 4
4. Count lower and upper bound scores for candidate grid cells.
5. Prune candidates based on axiom 5

First of all the grid is created given a number of axes for every dimension and using the mean data point. After that we create a dataframe given a set of points and we apply the grid. Later on for every cell the number of the points in the cell are counted using Spark's distributivity. Based on the previous counts, the lower and upper bound scores are computed. The next step is the implementation of the filter algorithm. The points and their coordinates are distributed but the coordinates of the cells, their defined borders and their scores are kept in the driver node.

After the filtering phase the points in pruned cells are set to check their dominating score. For every point p only the points in partially dominated cells are checked. For instance, cell 33 points will be checked against cell 34 points and cell 43 points for dominance and the lower bound score of the cell is added to form the final score. Finally a sorting is applied on scored points and the top k are displayed.

Scoring Algorithm:

1. Pruned Grid Cells PG
2. initialize scored points dataframe SP
3. For every cell in PG
    a. For every point in cell
        i. check for dominance in partially dominated cells
        ii. add lower bound score of cell
    b. union scored cell points SP
4. Sort scored_points_df in dissenting
5. Show top k

Again for step 3, 4, 5 spark sql queries and functionalities are used.

3.3 Top-k Dominating Skyline points

For the third task we combined the approaches from 2.1 and 2.2 with some adjustments. The skyline is computed as in 2.1 but there are some changes on the calculation of top k dominating points. Let's start with the adjusted axioms and extra axioms.

Axiom 6.
Given a grid cell g, g.fs is its total point count of cells fully dominating g and are part of the skyline.

Axiom 7. (Adjusted axiom 4)
Cells with zero count, with zero skyline points and cells with g.fs < k are not accounted as cells that we must check.

The Filter algorithm stays the same but only on the third step the candidates are produced based on axiom 7. The Scoring Algorithm changes taking and checking only for skyline points.

Adjusted Scoring Algorithm:

6. Pruned Grid Cells PG, Skyline S
7. initialize scored points dataframe SP
8. For every cell in PG
    a. For every point in cell that are also in S
        i. check for dominance in partially dominated cells
        ii. add lower bound score of cell
    b. union scored cell points with SP
9. Sort scored_points_df in dissenting
10. Show top k

4.Results

4.1 Skyline Results

Plenty of tests concerning the number of cores and volume of data were conducted. However, in order to export the skyline line plots the data were tested on a specific number of cores. Also, besides the 2 dimensional data plot there are no runtimes for 10.000.000 data points, due to increased tardiness. For the 3 dimensional and 4 dimensional the 10.000.000 datasets where tested, but there are no runtimes excluded. The execution time of the skyline code for the 3 and 4 dimensional data, of 10.000.000 data points, exceeded 2 days for the Anticorrelated skyline which is the most costly distribution. Also the Normal distribution which is the second more expensive distribution needs 1,5-2 days to run on 10.000.000 data.

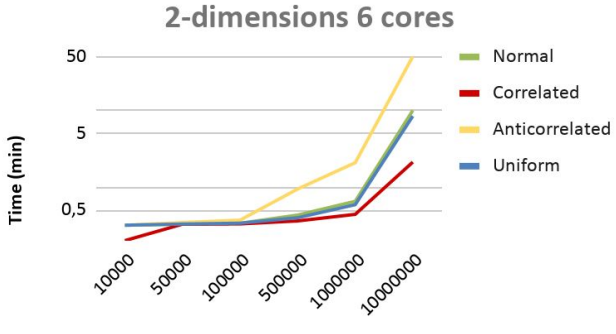

Figure 2: 2D 6-Cores time fluctuation changing dataset size

The above line plot represents the relation between each data distribution with their execution runtimes. The x-axis consists of the volume of data, whereas the y-axis consists of the time it took the code to be executed. It is obvious that the most time consuming distribution of all is the Anticorrelated one. Moreover it is apparent that as the volume of data increases and exceeds 100.000 the code runtime for the Anticorrelated data is on a continuous upward trend. Especially after 1.000.000 data points all distribution runtimes are augmented. The correlated distribution has the lowest runtimes, while the Uniform and Normal distributions follow a similar increase in their execution times, with the Uniform distribution having slightly lower execution times than Normal.

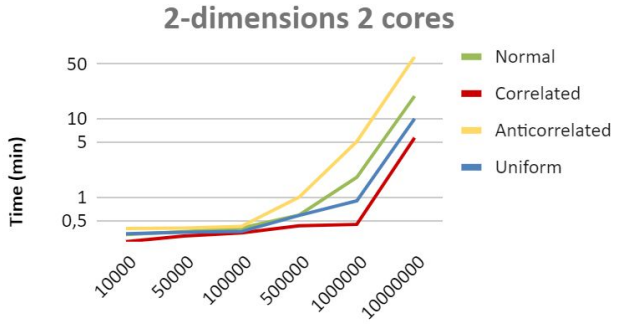

Figure 3: 2D 2-Cores time fluctuation changing dataset size

Similarly to the 2 dimensional runtimes on 6 cores the Anticorrelated distribution is the most time consuming and the Correlated distribution is the least time consuming. However in contrast to the 6 core runtimes the 2 core runtimes are significantly increased (especially for the Normal and Uniform datasets) as the volume of data exceeds 500.000 . In the above diagram it is clearly visible that the lowest execution time belongs to the Correlated distribution, which follows low time rates up until 1.000.000, while the Uniform distribution comes second and the Normal comes third.

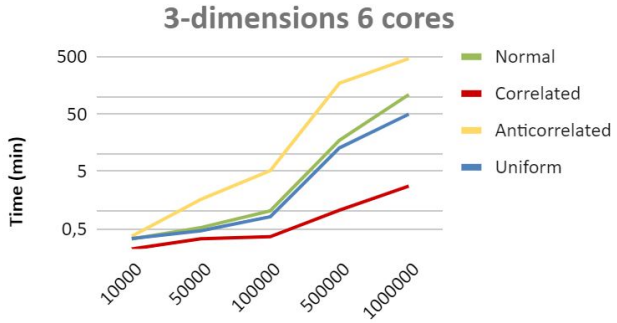

Figure 4 : 3D 6-Cores time fluctuation changing dataset size

On 3 dimensional data the runtimes differentiate from 2 dimensional data. Again the Anticorrelated distribution has the largest execution times, but in contrast to the times exported form 2 dimensions, increased times are noticed even on the lowest dataset (10.000), and the runtimes accelerate as the number of data grows. The lowest execution time is observed on the Correlated distribution, which starts having a rising course above 100.000 points. No matter how much the data increases the Correlated runtimes remain the lowest. Moreover the uniform and Normal distribution runtimes increase similarly, with Normal distribution slightly surpassing Uniform.

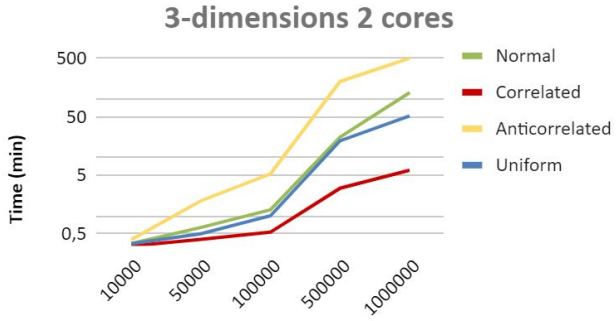

Figure 5 : 3D 2-Cores time fluctuation changing dataset size

Running the 3 dimensional data in 2 cores, the execution time of the code grows. The differences between the 6 and 2 cores are not visible on smaller datasets but when the data get bigger the times rise. Compared to the 6 core , the 2 core execution seems to increase the runtimes of Normal and Uniform distributions as well as the Correlated one. The correlated data showcase a major increase from 100.000 points and above which stabilizes, and follows a gradually growing course after 500.000 points. Again the Anticorrelated distribution is the costliest one, in terms of time.

In smaller datasets the differences between the 6 and 2 core runtimes are not that obvious. As the number of data points rise, the runtime gap between the distributions grows. This observation is quite reasonable, as more cores participating in the computation equals less runtime. Also increased data points equals more computations and therefore make the Skyline computation more computationally expensive.

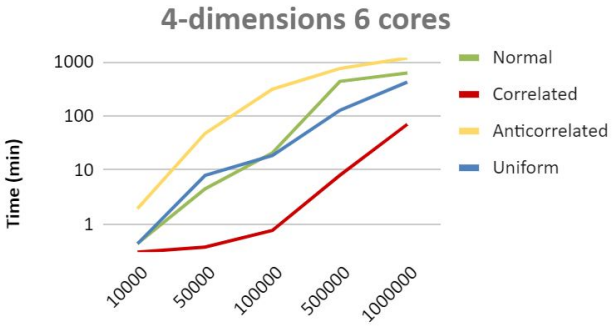

Figure 6 : 4D 6-Cores time fluctuation changing dataset size

Due to the fact that 4 dimensional data are demanding, and so the runtime of the skyline code takes longer to be executed, for the 4 dimensional data the skyline code ran only on 6 cores.

What we conclude from all executions is that the Anticorrelated distribution is the costliest of all, in terms of time. The previous observation is understandable if we take into consideration the values of each dimension on the Anticorrelated datasets (there are plenty points that are not dominated). It is the distribution with the most Skyline points, so more points are part of the local Skylines, in each partition, and more comparisons need to be made. Moreover more points are collected as Skyline points on the driver, therefore it is more computationally expensive. In contrast, the Correlated distribution contains few points with minimum values in each dimension, so there aren't many data points that don't get dominated. Thus, it computes the Skyline faster. The Normal and Uniform correlations are somewhere in between the two previous ones. Uniform has fewer points, compared to Normal distribution, that do not get dominated, but is still more expensive than the Correlated skyline computation. Also the data volume, number of cores, and the range of the dimensions directly affect the computational time

It is understandable that as the number of dimensions and the volume of data increase, value comparisons increase and so do the runtimes.

4.2 Top-k Dominating points

The main goal of the experiments that were held, was to test the limits of the algorithm and finetune certain parameters. It is evident that the task of finding the top-k dominating points is a computational expensive task and its complexity increases in an exponential manner. In the first experiment that was held, we are exploring the time that the algorithm needs to come to a conclusion and display the top k dominating points and how the algorithm behaves changing the dataset size, distribution and grid

size.

Figure 7 : 2D time fluctuation changing dataset size

In the sub-experiment of figure X we do not change the dimensions, the resources, the number of grid cells and the number of the top k dominating points. Clearly there is a gradual increase in time as the dataset size increases. As expected, in general, the algorithm has a hard time when it is applied in a dataset with Anti-Correlated distribution, outlier to this fact is when the dataset with 500 thousand points is used maybe because the algorithm was lucky on the splitting of points in the grid cells.

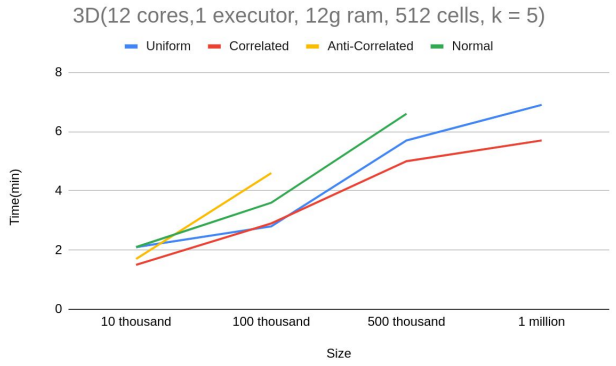

Figure 8: 3D time fluctuation changing dataset size

As in the previous sub-experiment we do not change the dimensions, the resources, the number of grid cells and the number of the top k dominating points. In figure X it is clear that the change in distribution from the first sub-experiment has caused the algorithm to fail on Anti-Correlated distribution when we increased the dataset size from 100 thousand to 500 thousands and on Normal dIstribution when we increase the dataset site from 500 thousand to 1 million. Again as expected that algorithm performs well when applied on uniform and correlated distributions.

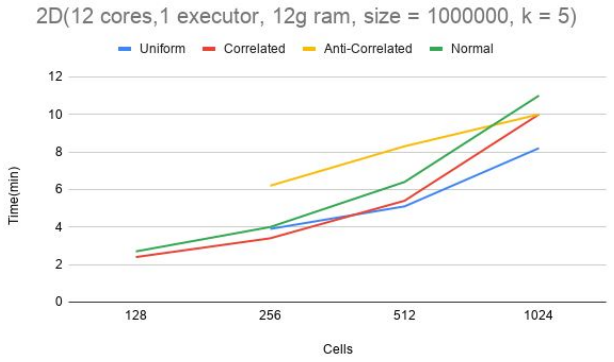

Figure 9: 2D cells fine tuning

In the experiment on the figure 6 we are trying to finetune the number of cells in the grid that is applied on a dataset, so for this reason we do not change the size of the dataset, the dimensions, the resources and the number of the top k dominating points. As we can see from figure 6 using a bigger grid cell doesn't mean better results and fine tuning the number of grid cells will increase the performance of the algorithm.

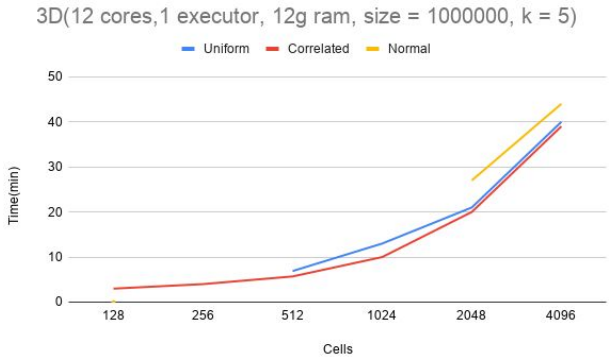

Figure 10: 3D cells fine tuning

In the figure 7 we can see that the algorithm fails on Anti-Correlated distribution and has a hard time on Normal distribution, needing a big grid with 2048 cells. Again the algorithm operates smoothly for Correlated distribution and has

some difficulties when is applied on Uniform distribution

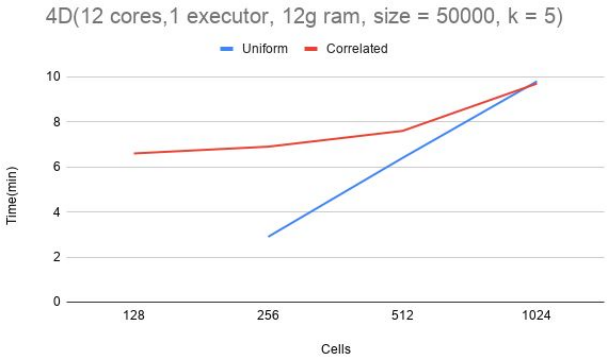

Figure 11: 4D cells fine tuning

For the experiment on figure 8, we decided to use a smaller dataset because the algorithm performance has decreased significantly on datasets with four dimensions. In figure 8 it is apparent that the algorithm fails to be applied in Anti-Correlated and Normal distributions at least using the system resources discussed on 1.2.

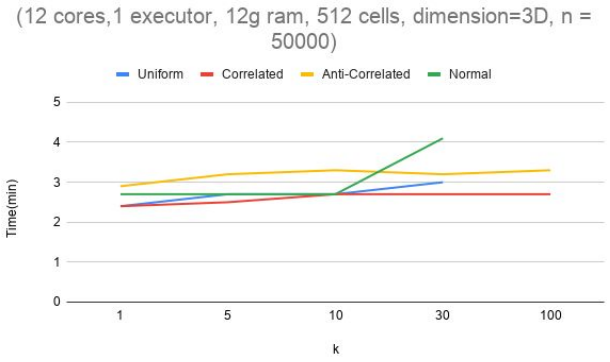

Figure 12: time fluctuation changing k

In the next experiment for task 2 we are trying to see how the time fluctuates changing the number of top k dominating points that the algorithm returns. For this experiment we don't change the dimensions, the resources the size of the grid and the dataset size. As you can notice the algorithm behaves very well for Correlated and Anti-Corrrelated distributions but fails on Uniform distribution and the reason is that too much cells are created if we use less grid cells it will succeed because the average number of points in a grid cell is increased, therefore the filtering and the pruning will exclude more cells from candidates.

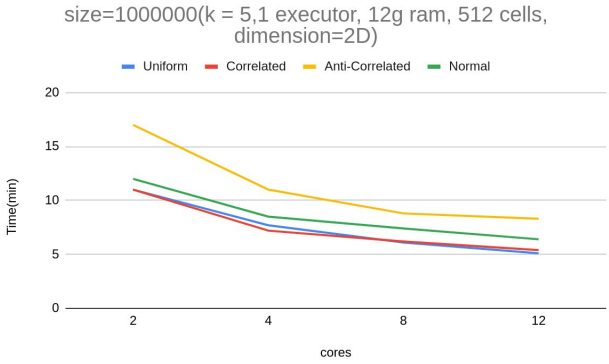

Figure 13: time fluctuation changing cores

For the last experiment we decided to test how good the algorithm for task 2 is parallelized. As you can see from figure 13 for all distributions we notice a decrease in time as the number of cores increases. This is a proof that the algorithm takes advantage of the extra cores and if we had a chance to give him even more, possibly we would see better times.

You can find the commands to run the algorithm for task 2 (in standalone Spark mode) in useful_commands.txt.

4.3 Top-k Dominating Skyline points Results

After the global skyline of each distribution was computed, the dominance score of the Top-K Skyline points was calculated. The diagrams that follow represent the time fluctuation as the number of data increases. The experiments that were conducted, concern the execution of the code with parameters such as k = 5 or k = 10 and cells checked 64 for the 2 and 3 dimensional data, and 16 for the 4 dimensional data.

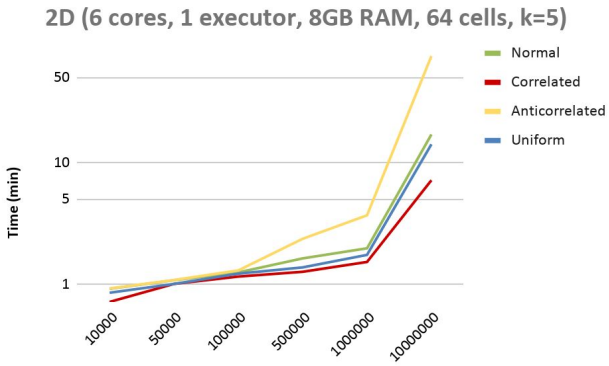

Figure 14: 2D time fluctuation changing dataset size

For 2 dimensions all distributions were tested from 10.000 up to 10.000.000 data points. All the executions were performed for 64 cells and the top 5 most dominating skyline points were extracted. It is clear that for a low number of data points all distributions have decreased execution times. As the volume of data increases and exceeds 1.000.000 the correlated distribution runtimes

increase and outdo the runtimes of the other distributions. Normal and Uniform data follow similar time patterns, while Correlated data are overall the lowest of all.

Figure 15: 3D time fluctuation changing dataset size

As the dimensions of data increase, so do the runtimes. For the 3 dimensional datasets, the top 10 most dominant Skyline points were generated, with 64 cells checked. It is noticeable that the execution of the code with the Anticorrelated data is the most time consuming. The Normal and Uniform distributions are quite similar in terms of execution time, with the Uniform being more expensive from 10.000 up until 100.000 points. Again the runtimes for the Correlated data note a gradual increase which however is reduced compared to the rest.

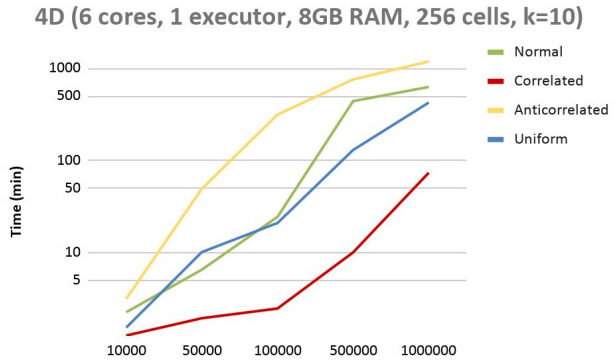

Figure 16: 4D time fluctuation changing dataset size

On the 4 dimensional data execution times reach 1200 minutes for the most time consuming distribution which is the Anticorrelated one. Thus the number of cells checked is 16 and the 10 Skyline points who are more dominant are returned. From the line plots of the above diagram it is obvious that besides the Anticorrelated, which is the most time consuming, and the Correlated , which is the least time consuming, distributions, the Normal and Uniform ones follow pequliar time patterns which alternate according to the volume of data.

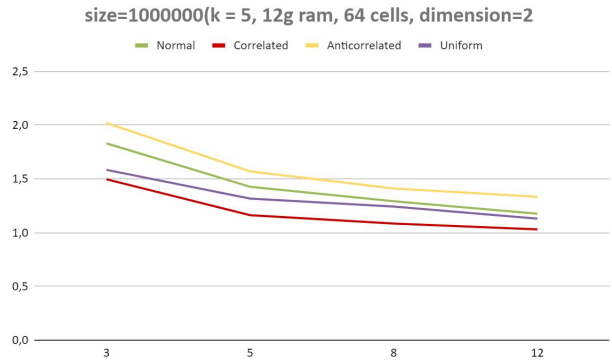

Figure 17: 2D time fluctuation changing cores

As it is apparent from the above diagram, as the number of cores increases the execution time of the program decreases. Again the Anticorrelated distribution has the highest runtimes while the Correlated one has the lowest. Also we should point out that the times concern only the execution of the 3rd task. In a system of multiple clusters we would have probably witnessed way less runtimes. Also this diagram proves that parallel execution becomes more efficient on bigger numbers of cores. The code takes advantage of any extra core given.

Overall, it is explicit that the Anticorrelated distribution is the costliest of all on every executed task and dimension. Respectively, the Correlated distribution is the least time consuming. Moreover the runtime of the code is affected by the number of cores participating in the execution as well as the memory capacity. If the code was executed on a cluster of computers then the execution times would have been drastically reduced. The execution of the Skyline and Top-k dominance scores on local machines isn't representative as the volume of data isn't Big enough to be considered Big Data and a single machine seeks to perform the function of several parallel machines each with their own Cores and memory storage. Therefore the code developed, on local computers, is useful to understand the parallel function of Spark and the results offer insight on the capabilities that it provides, however they are not representative of the powerful capabilities it offers.

KEYWORDS

Top-k dominating queries, Skyline, Dominance, Spark, Scala, Big Data

APPENDIX

GitHub: https://github.com/vlavrent/BigData

Drive:
https://drive.google.com/drive/folders/1oeqs45yz3BP4VJlbKz5KlaWafC2x4RES?usp=sharing

REFERENCES

[1] Jan Chomicki,Parke Godfrey, Jarek Gryz, Dongming Liang. (2013). Paper Skyline with Presorting. Computer Science, York University, Toronto, ON, Canada.

[2] ILARIA BARTOLINI, PAOLO CIACCIA, and MARCO PATELLA. Thesis. Efficient Sort-Based Skyline Evaluation, (2008)

[3] Yiu, M.L. and Mamoulis, N., 2009. Multi-dimensional top-k dominating queries. The VLDB Journal, 18(3), pp.695-718.

[4] Eleftherios Tiakas, Apostolos N. Papadopoulos and Yannis Manolopoulos. Top-k Dominating Queries: a Survey. Department of Informatics Aristotle University