Comparing the performance of graph analysis algorithms using Apache Flink Gelly and Apache Spark GraphX

Mohamed Gabr <mohgab@kth.se>
Óttar Guðmundsson <ottarg@kth.se>

Abstract: In this research, the graph processing libraries Gelly and GraphX are compared regarding the execution time of some algorithms processing batches of graph data. We transform open source network data into three undirected, connected graphs that vary in sizes. Three different algorithms, Connected Components, Graph Coloring, and PageRank, are used by both libraries and executed on the datasets. The time before and after execution is saved and their differences used as a benchmark. The results show that GraphX outperforms Gelly for all algorithms on all datasets, refuting results of past researches on a similar topic. However, our results hint that Flink manages memory better than Spark as more iterations on the algorithms are used, Spark runs out of memory. Tuning the properties of Spark can be a solution to this issue. Furthermore, we suggest future work on benchmarking other aspects of these libraries such as memory and CPU.

Table of contents

Preface

When the authors of this research, Mohamed and Óttar, grouped up for a research project they were both eager to ask each other about their origins. As Mohamed found out that Óttar was from Iceland, he instantly sent him a link from the social network Facebook about his friend that lives in Iceland. Unfortunately, Óttar did not know his friend, but he did know several of his mutual friends. Thus, they just found out about a connection between Egypt and Iceland through at least two different persons.

With modern technology, connections between humans and how they are connected have been easier to map than before. Graphs can structure such connections, which shows if there exists a connection between two objects and if the connection is one way or bi-directional.

Graph theory goes back up to the year 1736 when Euler wanted to see the most of Köningsberg but had to pass seven bridges to see the whole downtown area. He was in a hurry, so he only wanted to cross each bridge one time. The problem was later found out to be impossible to solve, creating the fundamentals of graph theory.

As both authors are pursuing an M.Sc degree at KTH in Software Engineering of Distributed systems, they also share a mutual interest in data science (another mutual connection!). Graph processing can be computationally complex based on the data and does describe multiple connections between objects. Thus researching a graph processing problem was not only motivating for them but highly relevant to their studies and interests. We hope that this research will benefit KTH in the future as both libraries, Gelly and GraphX, are used in courses such as Data Mining and Data-Intensive Computing which are a part of the data science track.

Introduction

The world is producing data more than ever before. Analyzing this massive amount of information interests the academic and the corporate field. Such material can provide valuable insights for companies and useful results for researchers. Therefore, developing efficient tools to perform these tasks is paramount in the field of Big Data. Information comes in different forms and structures; graphs are one of them. It is a data structure composed of nodes that are connected between each other by edges.

Social networks use graph data to represent their users as nodes and their connections to other users as edges. When a user is suggested, new friends or interesting topics, some process on the graph has been conducted, noticing that the two nodes have mutual connections through one or more edges, or the nodes share similar attributes [1]. Other interesting topics, like fraud detection and epidemiology researches, also involves the use of graphs. [2]

In recent years, big data graphs and relations between their data points has been growing at an enormous speed [3]. The demand for specialized tools to analyze these graphs has led to the development and release of several frameworks of graph processing libraries, such as GraphX in Spark and Gelly in Flink.

Nevertheless, it is challenging to choose the appropriate one for a particular task; the community lacks comparisons between these tools. Our project aims to benchmark GraphX and Gelly regarding runtime on diverse tasks with varying complexities. It intends to help future users picking the tool that serves their problem the best and hence saving their time. Additionally, by using the right tool when it comes to its actual production use will save computational resources.

Therefore, to determine which library is more suitable for processing batches of graph data concerning execution time, this study aims to research the performance on both of them by applying graph algorithms on three different datasets and benchmarking time taken to initialize the graphs, perform the computations and collecting the results.

Theoretical framework

Comparison between GraphX and Gelly, and their utilization on hardware has been researched, but choosing which one to pick when it comes to graph processing for projects that rely on graph algorithms performance and the time it takes to execute them requires further investigation.

Both Spark and Flink aim to achieve similar goals with a similar purpose, so a comparison between them has been conducted. Shelan Perera et al. [4] compared Spark and Flink in terms of runtime, CPU, and memory usage for the batch as well as stream processing. They reported that Flink clearly outperformed Spark in runtime. The CPU stress and the disk utilization test also favored Flink.

In contrast, O. Marcu et al.[5] found that there was no clear winner according to the authors. For them, Spark is faster for bigger graphs while Flink better for smaller graphs. They believe that the reasons for such differences are the JVM memory management, the pipeline execution nature of Flink, and the ease of optimization in the latter (done automatically) compared to Spark.

Diego García-Gil et al. [6] have shown that the machine learning libraries for both Spark and Flink, that MLlib outperforms FlinkML in scalability and speed, hinting that additional packages for Spark might outperform Flink due to more contributors. FlinkML can still achieve good results with low latency but does require external third-party tools as of this moment.

Additional research on analyzing the streaming capabilities of this two software (plus Storm) has been done [7] but solely to mimic real-world production scenarios, only testing for the latency of max the throughput. This comparison did not include the execution time of the processes as data increased nor tested out for different types of graph processing algorithms.

We can conclude from the previous articles and research papers that there is no clear answer to whether Spark performs better than Flink concerning the execution time when processing algorithms on graphs and how it changes as the graph size increases. Therefore, we believe that this topic needs further investigation. Benchmarking and measurement techniques, as well as significant parameters, are motivated by the previous literature.

Research questions, hypotheses

The question that we will try to answer, which of the two chosen libraries outperforms the other in graph processing, is there a clear winner? We hypothesize that Spark should perform better using GraphX for any of the tested algorithms when all of the data is already stored and available. We define better performance based on the execution time of the algorithms.

Research Methodology

Data

The graph data processed by the libraries were all undirected and fully connected networks. Thus, connections between nodes are bidirectional, and all nodes have at least one connection to another node, making the problem solving more computationally heavy. Three different datasets were used in the research ranging from a hundred to ten thousand nodes, as seen in Table 1, to test for performance in respect to the size of the graph.

All the datasets used in this project describe human to a human connection from a public repository [8]. However, they describe different types of information; the smallest one depicts connections between a group of jazz musicians, the second pictures mail interchanges among members of a University and the third a network of secure information interchanges between users. The type of information does not affect the process of the libraries in any way because nodes do not hold any specific properties. Furthermore, by processing real-world data, we emphasize the capabilities of both libraries being able to solve real-world problems.

For every line in all datasets, white spaces and tabs were trimmed from the beginning to the end. A single white space separated the connection from one node to the other, thus importing data into libraries was reasonably straightforward.

Edges are imported as a bidirectional connection, making the graph undirected. Even though the datasets are defined as undirected, the dataset only contains half the connections. It is intentional with the aim to reduce the data size by half and to speed up the graph structure.

The datasets were further investigated with the open source graph-analysis tool in Python, NetworkX [9]. The metadata of the dataset should be matched with information from the analysis tool to confirm that the graph is fully connected and matches the metadata description. A few more exciting properties were found such as density and average degree, which we will reflect on later in this research.

Table 1: Statistical information about each graphs properties

Variables/Datasets	Jazz Musicians	U. Rovira i Virgili	Pretty Good Privacy
Number of nodes	198	1,133	10,680
Number of edges	2,742	5,451	24,316
Max degree	100	71	205
Min degree	1	1	1
Average degree	27,697	9.6222	4.5536
Density	0.14059375	0.00850021	0.0004264
Triangle count	17,899	5,343	54,788

Algorithms

The benchmarking involves three different algorithms with varying complexities: Graph Coloring, Connected Components, and PageRank. Both libraries provide PageRank and Connected Components algorithms.

We adopt a common damping factor of 0.0001, to have a legitimate comparison between the two libraries for PageRank. The method *staticPageRank* used in Spark allows us to set the number of iterations. The only one PageRank method in Flink takes as an argument the number of iterations. The user must specify the maximum number of iterations for the Connected Components method, not the actual number, for both Flink and Spark. However, the libraries do not provide the graph coloring algorithm. We show its pseudo-code before explaining the code in Figure 1.

```
function ColorReductionFast(G = (V, E), Δ, maxIter)
    // Step 1
    n = |V |
    for v =1 to Range(n) do
        color(v) = v
    end for

    // Step 2
    for i = 1 to maxIter do
        for each vertex v:
            color(v) = min({1, ..., Δ + 1} \ {color(u) | (u, v) ∈E})
            such that color(u) < Δ +1  && color (v) > (Δ + 1)
        end for
    end for
end function
```

Figure 1: Pseudo-code for Graph Coloring algorithm, implemented in both Spark and Flink

The first step is to define a different color to each node. Here every color is a number, and the span of the definition should be of [1; noNodes]. The second step is, for each vertex, choose the minimum color number of all neighbors if the color of the current vertex is bigger than delta+1 and the color of the neighbor is smaller than delta+1. This step runs as much as the user wants. The more iterations, the more the result converges. We wrote this algorithm in Spark and Flink, and we took care to have a very similar implementation. In both Spark and Flink, a for loop is responsible for the first step and is not part of the benchmarking, since it is considered part of the initialization.

For Spark, the implementation of the second step is done using two different GraphX in-built methods. The first one is *aggregateMessages*, that takes every vertex, and sends to all its neighbors a specific value. Messages that do not satisfy the condition defined in the code are not sent. In this same method, one must define a function to reduce the messages that arrive in the vertex. Here, it is taking the minimum. The second part is outerJoinVertices, which is accountable of updating the graph with the new color values; it checks if a vertex received any messages first (by checking if the reduce gave -23 or not, where -23 means no messages arrived). If it is the case, it updates the value of the given vertex. This whole part of the code runs as much as the user wants, given that the number of iterations is defined.

The implementation of the second step is done using Gelly Scatter-Gather technique in Flink. We believe Scatter-Gather is the most similar method to the implementation in Spark. The first part of Scatter-Gather, Scatter, sends a message with a value to all neighbors for each vertex, the color in our case. The second part, Gather, goes through each message, we define here the method to aggregate them and get the minimum. Messages that do not satisfy the condition defined in the code are not aggregated. We manually check if a message is received with a boolean. After checking all the messages and aggregating them, the value of the current vertex is updated if a message is received. The number of iterations is passed as an argument.

We intentionally choose to have at least one built-in algorithm and one that is written by ourselves, to compare the performance with ready-made implementations as well as with algorithms that use the tools given to build one.

Measurements

Execution time will be measured in nanoseconds instead of milliseconds to provide more precision for the comparison. Java includes two methods for measuring time, System.nanoTime and System.getCurrentMillis, where the former one returns 1/1000000 of a second while the latter around half of it [10]. Using nanoTime can in some cases produce inconsistent results, [11] taking more time to compute and not being thread-safe, and is thus not recommended for scalability or saving multiple checkpoints in a script. Our measurements will only be checked before and after the algorithm on a single machine, allowing more precision without being affected or affecting the algorithmic computation.

The timing is done by instantiating a new long variable before and after each function, that represents the daytime clock of the computer. Afterward, the difference between the two is calculated by subtracting the time before from the time after. The time difference was then saved to a table according to the algorithm, dataset and the library used as can be seen in the appendix. The final results in our tables will be compared, and if there are unexpected differences, an attempt will be made to explain them. To assure that the measurements are trustworthy the significance of the timing of both libraries will be computed.

The independent variable of this experience are the libraries, Gelly and GraphX while the dependent variable will be the execution time of the algorithms at different stages of the process. Three different dependant time variables will be considered, graph initialization, processing the algorithms on the graph, and collecting the results and printing out in the console. Different compilers and development environments could potentially affect the runtime even though the written code was simply imported between computers. Thus, the IDE and the compiler were identified as control variables, having the same version.

Environment

The selected algorithms will run on the two mainstream computers using the two libraries. These two chosen computers are the Retina, 13-inch, Early 2015 MacBook Pro and the T450s IBM/Lenovo Thinkpad. There are two reasons for these picks, the former being that these computers are ubiquitous [12] and there was no budget for additional expenses for this research. On the second computer, a virtual Ubuntu environment is hosted to run the experiments to test out if the same results can be replicated on a virtual machine and to ease other researchers producing the same results. The specs of the computers and the simulated environment used can be found in Table 2.

Table 2: Specs of both computers used and the software used for the environment

	Computer 1	Computer 2	Virtual environment (runs on Computer 2)
Model Name	MacBook Pro (Retina, 13-inch, Early 2015)	Thinkpad T450S	-
Processor	2.7GHz dual-core Intel Core i5 processor (Turbo Boost up to 3.1GHz) with 3MB shared L3 cache	Intel® Core™ i5-5300U Processor (3M Cache, 2.30 GHz)	Intel® Core™ i5-5300U Processor (3M Cache, 2.30 GHz)
Graphics	Intel Iris Graphics 6100	Intel HD Graphics 5500	
Memory	8GB of 1866MHz LPDDR3 onboard memory	12GB DDR3L 1600 MHz (1 DIMM)	4596 mb
Storage	256GB PCIe-based flash storage	500GB Hard Disk Drive, 7200 rpm	256GB Hard Disk Drive, 7200 rpm
Operating System	macOS High Sierra Version 10.13.6	Windows 10 Pro	Ubuntu 18.04.01 LTS (Bionic Beaver)
IDE	IntelliJ IDEA 2018.2.3 (Community Edition) Build #IC-182.4323.46, built on September 3, 2018 JRE: 1.8.0_152-release-1248-b8 x86_64 JVM: OpenJDK 64-Bit Server VM by JetBrains s.r.o macOS 10.13.6	-	IntelliJ IDEA 2018.2.3 (Community Edition) Build #IC-182.4323.46, built on September 3, 2018 JRE: 1.8.0_152-release-1248-b8 amd64 JVM: OpenJDK 64-Bit Server VM by JetBrains s.r.o
Compiler	sbt	-	sbt
JVM Maximum Heap Size	1536 MB	-	

Results and Analysis

The runtime of all three different algorithms proposed was tested successfully using the Spark GraphX and Flink Gelly libraries. The benchmarked runtime does not include importing the graph from storage and includes one action on the collection and not only transformations to materialize the results.

There is no increase in runtime for the Connected Components algorithm in respect to the number of iterations because it represents the maximum. If the results converge, the algorithm stops iterating as seen in Figure 2.

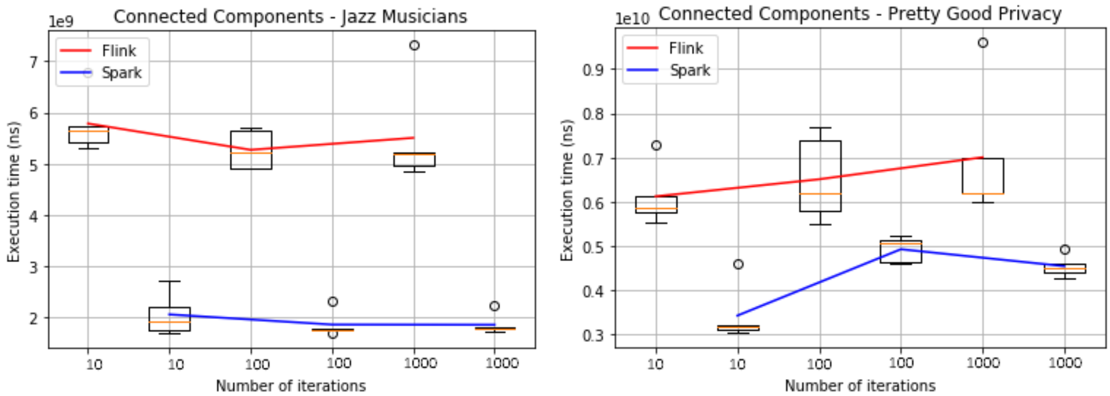


Figure 2: Spark outperforms Flink in the inbuilt algorithm Connected Components.

The Graph Coloring algorithm is a self-made implementation as explained in the algorithmic section, unlike the two others that are inbuilt with the library. We took care of using the most similar technique to apply graph coloring. Nevertheless, there is a slight difference, Flink scatter-gather algorithm stops iterating after convergence, which is not the case for Spark. It is the reason behind the inexistent rise in runtime for Flink Graph Coloring as seen in Figure 3.

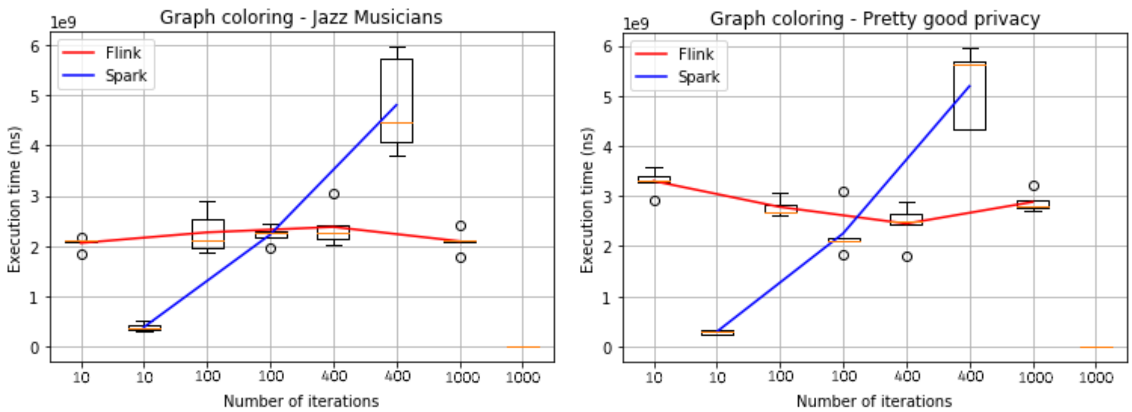


Figure 3: Graph coloring converges automatically in Flink, while Spark keep iterating before running out of memory.

Spark performed noticeably faster than Flink for PageRank on all datasets. However, as the number of iterations of the algorithm increased, Spark ran out of memory as seen in Figure 4. The Spark garbage collector recovered a minimal amount of memory at every iteration, never reaching 1000 iterations.

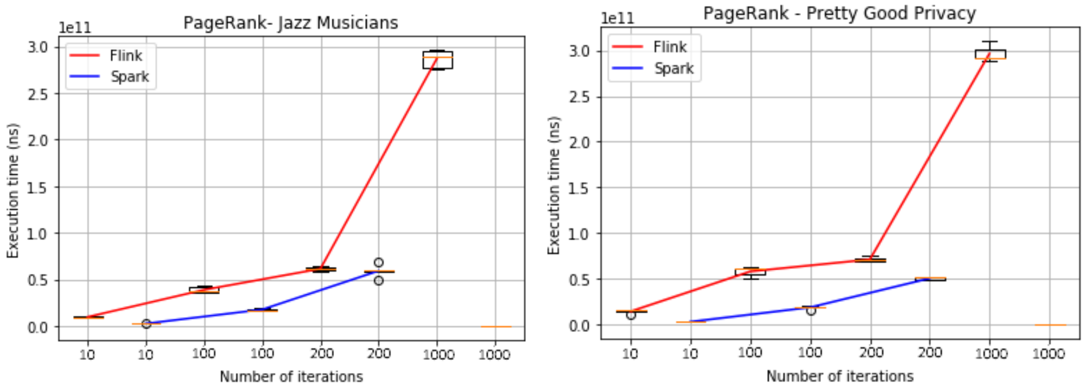


Figure 4: Even though Spark outperforms Flink once again, the time taken increases linearly for both libraries. However, Spark does not handle the memory as efficiently.

The t-value table displays the significance/t-scores (add a footer about what is significance) of the discrepancy between the results in Spark and Flink for every algorithm, with different numbers of iterations. It answers the following hypothesis: “Spark is faster than Flink for a specific number of iterations, for a particular algorithm.” The t-value table shows that it is the case of both milliseconds and nanoseconds, all values are superior to 2.61 which gives a p-value well below .025, equivalent to a confidence level superior to 95%. The hypothesis we established earlier in the project for batch processing is thus validated.

It is also valid in seconds; except for one outlier (Jazz Musicians - PageRank - 200 iterations), the confidence level is superior to 95%. It means that for the same scale of graphs we tested, the choice of the library is essential in respect to runtime even if seconds is the unit of measurement.

Note that the t-values in Graph Coloring benchmarks are not relevant in our hypothesis, due to the nature of implementation of the algorithms, as explained above in this section. Despite this fact, Spark still outperforms Flink for up to 10 iterations.

It is also clearly seen that the number of iterations influences the runtime increase more than the graph size increase. We define the graph size by its number of vertices, although there are multiple ways to describe it, like the number of edges, average degree, and others found in Table 1. The runtime increase in respect to the number of iterations is linear in our logarithmic scale graph for graph coloring, hence exponential.

Discussion

We can infer from the results that Spark is the clear winner, which contradicts most of the literature we studied. The majority agreed that Flink outperforms Spark on smaller graphs. Numerous reasons can be behind this opposition. The two libraries are evolving rapidly, and an inspection made four years ago can already be judged obsolete.

Most of the previous benchmarks are optimizing the settings to obtain a “fairer” comparison. Our aim is different; we believe that by comparing the two without tuning is another way of benchmarking. There are countless users in the community that would like to know which one is faster with the out-of-box settings.

Even though Spark is faster in our analysis, the in-memory management in Flink proves to be efficient; Flink writes off-heap/on-disk when it goes out of memory while Spark crashes the node in the same situation by default. We witnessed this in our tests; Spark threw an out-of-memory error with PageRank and Graph Coloring set to 1000 iterations but not Flink. [13]

Although our paper is about comparing the two libraries regarding performance, we must emphasize that Spark has a more significant community than Flink which makes the implementation easier. One finds an answer to a question for Spark quicker than for Flink.

There are multiple limitations to our benchmarking. It does not operate on multiple nodes, thus not testing how well Flink and Spark are scaling and network usage efficiency. We did not monitor the resource usage (memory, disk) which may be an essential factor for some users when choosing a suitable library.

The results and conclusions of this test are suitable for an ordinary user, who wants to run a processing algorithm on a mainstream personal machine without worrying about the hassle of tuning the parameters.

Conclusion

As big data grows unsurprisingly fast, the community is in critical need for better and more powerful processing tools. Graph processing is no exception, as more and more data gathering situations involve this type of collections. We aimed to investigate two of the most prominent tools to analyze graphs: Graphx in Spark and Gelly in Flink.

In this research, we concluded that the graph processing library of Spark, GraphX, outperforms Gelly regarding execution time of three algorithms on three distinct datasets in all circumstances. Nevertheless, we discovered that Flink automatically manages memory better than Spark as the latter ran out of memory when iterations on specific algorithms increased. Optimizing particular parameters in Spark overcomes this issue, addressing the needs of future work on analyzing how well Spark can be tuned to rival Flink concerning performance and memory, using accredited benchmarking tools.

The definite conclusion is that Spark should be employed to process graphs, as long as the iterations do not deplete all the available memory. However, this is only true if one does not want to bother with configuring parameters. Not only is it faster regarding execution, but developing code for Spark is significantly more comfortable due to a larger community and an abundance of online code samples.

References

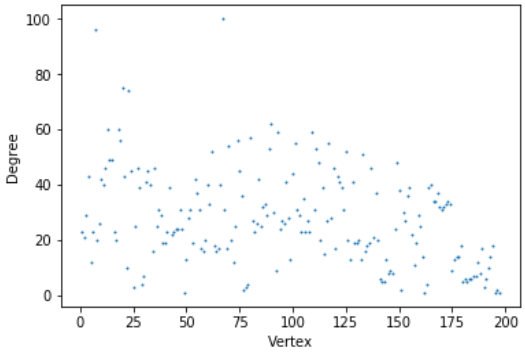
[1] "An Introduction to Graph Theory and Network Analysis (with Python codes)," *Analytics Vidhya*, 19-Apr-2018. .
[2] G. Sadowksi and P. Rathle, "Fraud Detection: Discovering Connections with Graph Databases," p. 12.
[3] "The growing graph database space in 2018," *Information Age*, 11-Jan-2018. .
[4] S. Perera, A. Perera, and K. Hakimzadeh, "Reproducible Experiments for Comparing Apache Flink and Apache Spark on Public Clouds," *ArXiv161004493 Cs*, Oct. 2016.
[5] O.-C. Marcu, A. Costan, G. Antoniu, and M. S. Perez-Hernandez, "Spark Versus Flink: Understanding Performance in Big Data Analytics Frameworks," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, Taipei, Taiwan, 2016, pp. 433–442.
[6] "A comparison on scalability for batch big data processing on Apache Spark and Apache Flink | SpringerLink." [Online]. Available: https://link.springer.com/article/10.1186/s41044-016-0020-2. [Accessed: 03-Oct-2018].
[7] "Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming - IEEE Conference Publication." [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7530084. [Accessed: 03-Oct-2018].
[8] "KONECT - The Koblenz Network Collection," 02-Oct-2018. [Online]. Available: http://konect.uni-koblenz.de/networks/. [Accessed: 02-Oct-2018].
[9] "NetworkX — NetworkX," 02-Oct-2018. [Online]. Available: https://networkx.github.io/. [Accessed: 02-Oct-2018].
[10] "Java System.nanoTime() vs System.currentTimeMillis," *GeeksforGeeks*, 12-Apr-2018. .
[11] "Nanotrusting the Nanotime," 02-Oct-2018. [Online]. Available: https://shipilev.net/blog/2014/nanotrusting-nanotime/. [Accessed: 02-Oct-2018].
[12] D. A. July 30 and 2018 Laptops, "Best business laptops 2018: top laptops for work," *TechRadar Sweden*, 02-Oct-2018. [Online]. Available: https://www.techradar.com/news/best-business-laptops. [Accessed: 02-Oct-2018].
[13] "Apache Spark vs Apache Flink - 8 useful Things You Need To Know," *EDUCBA*, 05-Mar-2018. .

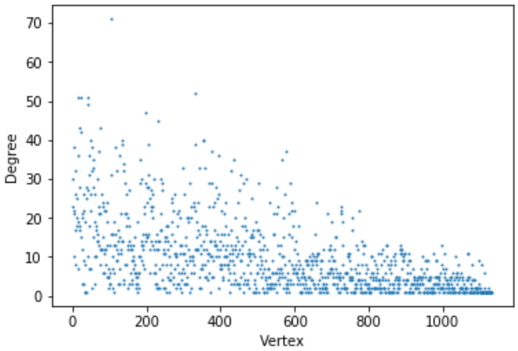
Appendix

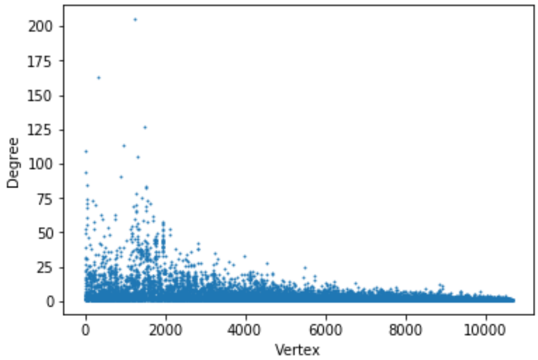
All measurements

Algorithm	**PageRank**					
Dataset	*Jazz Musicians*		*I rovira*		*Pretty Good Privacy*	
Max iterations	Flink	Spark	Flink	Spark	Flink	Spark
10	1.03E+10	2.23E+09	1.34E+10	3.48E+09	1.54E+10	2.59E+09
10	8.81E+09	2.21E+09	9.60E+09	2.20E+09	1.49E+10	2.48E+09
10	9.32E+09	2.36E+09	9.86E+09	2.26E+09	1.49E+10	2.70E+09
10	9.16E+09	3.01E+09	1.71E+10	2.62E+09	1.37E+10	2.75E+09
10	1.03E+10	2.50E+09	1.13E+10	2.33E+09	1.08E+10	2.65E+09
100	3.52E+10	1.99E+10	4.06E+10	2.11E+10	6.21E+10	2.01E+10
100	3.69E+10	1.68E+10	4.36E+10	2.33E+10	6.06E+10	1.94E+10
100	4.34E+10	1.81E+10	4.07E+10	1.77E+10	6.08E+10	1.84E+10
100	3.55E+10	1.65E+10	4.09E+10	2.93E+10	5.45E+10	1.84E+10
100	4.25E+10	1.64E+10	4.11E+10	1.86E+10	5.08E+10	1.55E+10
200	6.23E+10	6.01E+10	7.46E+10	7.00E+10	7.01E+10	5.19E+10
200	5.90E+10	5.77E+10	7.20E+10	5.81E+10	7.47E+10	5.12E+10
200	6.15E+10	5.97E+10	6.97E+10	6.10E+10	6.90E+10	4.82E+10
200	6.42E+10	4.93E+10	6.46E+10	6.10E+10	6.88E+10	4.85E+10
200	5.93E+10	6.85E+10	6.41E+10	5.30E+10	7.17E+10	5.12E+10
1000	2.96E+11	Out of Memory	3.37E+11	Out of Memory	3.01E+11	Out of Memory
1000	2.76E+11	Out of Memory	3.35E+11	Out of Memory	2.91E+11	Out of Memory
1000	2.89E+11	Out of Memory	3.03E+11	Out of Memory	2.89E+11	Out of Memory
1000	2.77E+11	Out of Memory	3.21E+11	Out of Memory	2.91E+11	Out of Memory
1000	2.95E+11	Out of Memory	3.09E+11	Out of Memory	3.10E+11	Out of Memory

Degree distribution of datasets







t-values Table

	Jazz Musicians			Pretty Good Privacy			I Rovira		
	Difference Significance in ns			Difference Significance in ns			Difference Significance in ns		
No Iterations	**Graph Coloring**	**Connected Comps**	**PageRank**	**Graph Coloring**	**Connected Comps**	**PageRank**	**Graph Coloring**	**Connected Comps**	**PageRank**
10	265607.0946	261506.3005	500163.8292	403633.7775	163854.7829	569856.365			358983.6812
100	51372.53243	300015.7908	642126.6702	45666.18945	99436.32621	1081843.788			564734.7712
200 - 400	-145465.9076	-	51812.99018	-177378.6038	-	716957.1015		-	180585.6498
1000	Out of Mem	232476.8267	Out of Mem	Out of Mem	131221.8569	Out of Mem	Out of Mem		Out of Mem
	Difference Significance in ms			**Difference Significance in ms**			**Difference Significance in ms**		
10	265.6070946	261.5063005	500.1638292	403.6337775	163.8547829	569.856365			358.9836812
100	51.37253243	300.0157908	642.1266702	45.66618945	99.43632621	1081.843788			564.7347712
200 - 400	-145.4659076	-	51.81299018	-177.3786038	-	716.9571015		-	180.5856498
1000	Out of Mem	232.4768267	Out of Mem	Out of Mem	131.2218569	Out of Mem	Out of Mem		Out of Mem
	Difference Significance in s			**Difference Significance in s**			**Difference Significance in s**		
10	8.399233815	8.269555321	15.81656903	12.76402078	5.181543196	18.02044053			11.35206076
100	1.624542116	9.487332328	20.30582824	1.444091707	3.14445273	34.21090443			17.85848151
200 - 400	-4.600035899	-	1.638470614	-5.609203962	-	22.67217425		-	5.710619661
1000	Out of Mem	7.351562755	Out of Mem	Out of Mem	4.149599467	Out of Mem	Out of Mem		Out of Mem