Comparison between Python scaling frameworks Ray and Apache Spark for Big Data Analysis and Machine Learning

Konstantinos Tsokas
School of Electrical and Computer Engineering
National Technical University of Athens
Athens, Greece
konstantinos.tsokas@hotmail.gr

Ioanna Kioura
School of Electrical and Computer Engineering
National Technical University of Athens
Athens, Greece
kiouraioanna@gmail.com

***Abstract*—In the realm of distributed computing frameworks, Ray and Apache Spark stand out as powerful tools that facilitate the scaling of Python analytics code for Data Analytics, Machine Learning, and Artificial Intelligence workloads. Ray, an open-source unified computing framework, provides a robust ecosystem for scaling Python workloads. Apache Spark, a widely adopted and versatile distributed computing framework, is renowned for its ability to handle diverse analytics tasks. This research undertakes a comprehensive comparison between Ray and Apache Spark in performance and capabilities. We install and set up both of these systems utilizing remote resources. We then generate the necessary data that we will utilize for our comparison and load them into both systems. At last, we develop performance measurement code to measure the performance of Ray and Apache Spark across various machine learning and big data-related operations. These scripts explore individual tasks, such as graph operators and popular ML operations, catering to diverse cluster resources and input data sizes/types. Our research provides valuable insights into the relative strengths and weaknesses of Ray and Apache Spark, so that teams can make informed decisions based on meaningful statistics. More specifically, we come to the conclusion that Apache Spark is superior regarding the machine learning tasks that we tested it on, but Ray's speed is matchless in graph operations. Ray's disadvantage however is the large amount of memory it requires to operate. The outcomes contribute to a deeper understanding of each system's strengths and overall suitability for Python analytics workloads.**

***Index Terms*—Spark, Ray, Python, scaling, big data, machine learning**

I. Introduction

Presently, the significance of big data analytics and machine learning in contemporary data-driven applications is constantly increasing. As organizations grapple with ever-expanding datasets and complex analytical tasks, selecting the right framework becomes paramount. Apache Spark and Ray represent two distinct paradigms in the realm of distributed computing, with Spark being a well-established solution and Ray emerging as a promising contender. The motivation behind this research lies in addressing the practical needs of developers, data scientists, and analysts who face the challenge of selecting the most suitable framework for their specific use cases. Our decision to evaluate these frameworks across various tasks, diverse input data sizes, and variable node configurations aims to provide a holistic view of their performance characteristics. By benchmarking their capabilities under different conditions, we intend to offer nuanced insights that extend beyond generic comparisons. This research not only contributes valuable benchmarks for practitioners but also serves as a guide for researchers seeking to understand the trade-offs and advantages of these frameworks in real-world scenarios. Ultimately, our aim is to empower the broader community with informed choices when navigating the landscape of big data analytics and machine learning frameworks.

In this research paper, we delve into a comprehensive comparative analysis between Apache Spark and Ray, two prominent frameworks for handling big data analytics and machine learning tasks. Our investigation focuses on evaluating their performance across a spectrum of individual tasks, encompassing graph operators such as PageRank as well as popular machine learning operations including prediction and clustering. We produce codes implementing these algorithms and run various experiments based on them. In our investigation, the primary focus is placed on quantifying execution time and memory usage, with an emphasis on scalability and efficiency, rather than on assessing the accuracy of the processes.The experimentation involves varying the number of nodes/workers and manipulating input data sizes and types, providing a nuanced understanding of the scalability and efficiency of both Apache Spark and Ray in real-world scenarios. Through this study, we aim to contribute valuable insights that shed light on the relative strengths and weaknesses of these frameworks, aiding practitioners and researchers in making informed decisions for their specific big data and machine learning requirements.

II. Software and infrastructure utilized

For this research, we used specific infrastructure and software to support and carry out our experiments.

A. Okeanos-Knossos

Okeanos-Knossos is a cloud infrastructure service provided by the Greek Research and Technology Network (GRNET) for research purposes. It is provided to NTUA students for free. It offers Infrastructure-as-a-Service (IaaS) capabilities, allowing users to deploy and manage virtual machines, storage, and other cloud resources. The essence of IaaS lies in its provision of a virtualized computing infrastructure, eliminating concerns related to hardware maintenance, network intricacies, and software management. With okeanos-knossos, users gain direct and convenient control over their computing environment through a web browser interface. All in all, this platform facilitates the creation, administration, and interaction with Virtual Machines and Virtual Networks, providing a streamlined and accessible approach to computing resources without the burden of hardware and networking complexities.

B. Hadoop

Hadoop is an open-source framework designed for distributed storage and processing of large-scale data sets across clusters of commodity hardware. Developed by the Apache Software Foundation, Hadoop facilitates the processing of vast amounts of data using a programming model known as MapReduce. At its core, Hadoop consists of two fundamental components: Hadoop Distributed File System (HDFS) for distributed storage, and the MapReduce programming model for parallel data processing. Hadoop's distributed and fault-tolerant nature enables it to handle diverse types of data and scale horizontally to accommodate increasing data volumes. With its robust ecosystem of tools and utilities, including Apache Hive, Apache Pig, and Apache Spark, Hadoop has become a cornerstone in the field of big data analytics, providing organizations with the capability to derive valuable insights from massive datasets.

C. Apache YARN

Apache YARN (Yet Another Resource Negotiator) is a key component of the Apache Hadoop ecosystem, specifically designed to manage resources and schedule tasks within a Hadoop cluster. YARN serves as a resource management layer that allows multiple processing engines to coexist and share resources efficiently. Unlike its predecessor, the MapReduce-centric Hadoop 1.x architecture, YARN introduces a more flexible and scalable framework, enabling diverse workloads such as real-time processing, graph processing, and interactive queries. YARN's architecture separates resource management and job scheduling, providing a generalized platform for running various distributed applications. This decoupling enhances the overall resource utilization and makes Hadoop a more versatile and extensible framework, supporting the evolving needs of modern data processing and analytics workflows.

D. Apache Spark

Apache Spark stands as a powerful open-source distributed computing system designed for fast and versatile data processing. Renowned for its speed and ease of use, Spark extends beyond the capabilities of its predecessor, Hadoop MapReduce, by enabling in-memory data processing, which significantly enhances performance. With support for various programming languages such as Java, Scala, and Python, Apache Spark simplifies the development of complex data applications. Its advanced analytics engine facilitates the processing of large-scale datasets, while its built-in libraries cover diverse tasks like SQL queries, machine learning, graph analysis, and stream processing. Apache Spark excels in handling both batch and real-time data processing, making it a preferred choice for organizations dealing with diverse and substantial data workloads. Apache Spark serves as an exemplary scaling framework for big data analysis and machine learning, offering a versatile and efficient platform for processing vast datasets across distributed computing clusters. Its inherent ability to perform in-memory computations minimizes data transfer overhead, resulting in accelerated processing speeds. Spark's resilient distributed datasets (RDDs) provide fault tolerance, ensuring reliable handling of large-scale data across distributed nodes. Moreover, Spark seamlessly integrates with machine learning libraries, enabling the scalable execution of complex analytics and model training tasks. The ease of use and compatibility with diverse programming languages make Apache Spark a compelling choice for organizations seeking to scale their data analytics and machine learning workloads seamlessly.

E. Ray

Ray is an open-source unified computing framework designed to simplify the scalability of data analytics, machine learning, and artificial intelligence tasks in Python. Developed to address the challenges of scaling Python workloads, Ray enables users to efficiently process large-scale datasets. Its versatile features include Ray Datasets, a high-performance API tailored for robust data processing. Ray's strengths lie in its capacity to handle extensive datasets, making it well-suited for resource-intensive operations. Users benefit from the ease of scaling computations and managing virtual machines and networks with just a few clicks through their web browsers. Ray's popularity stems from its contribution to enhancing the scalability and efficiency of Python-based analytics and machine learning tasks.

III. System set up

A. Okeanos-Knossos

We connected to okeanos-knossos using our academic emails. As NTUA students, resources are available to us to begin our work. We enrolled in the course's project and created a set of cryptographic keys. Afterward, we registered the public key at okeanos-knossos. We then built three virtual machines on okeanos-knossos: a master and two workers. Each machine had:

- Ubuntu Server LTS 16.04 OS
- 4 CPUs
- 8GB RAM
- 30GB disk capacity

We connect to these remote machines using the ssh protocol. It was subsequently crucial to develop a public and a private network for our machines. For the creation of the public network, okeanos-knossos offers us a unique public IPv4 to connect to, which we attached to the machine we named master. Using okeanos-knossos's Manage Networks tab, we also developed the private network through which our three virtual machines communicate. For each machine, a private IP was set. Consecutively, we produced new sets of cryptographic keys from our master in order to contact it and the worker nodes.

Lastly, we opted to update and upgrade the virtual machines' operating system. The choice was made to ensure total compatibility with the software we utilized for our research. More specifically, we upgraded from Ubuntu 16.04 to Ubuntu 18.04, then to Ubuntu 20.04 and lastly to Ubuntu 22.04.

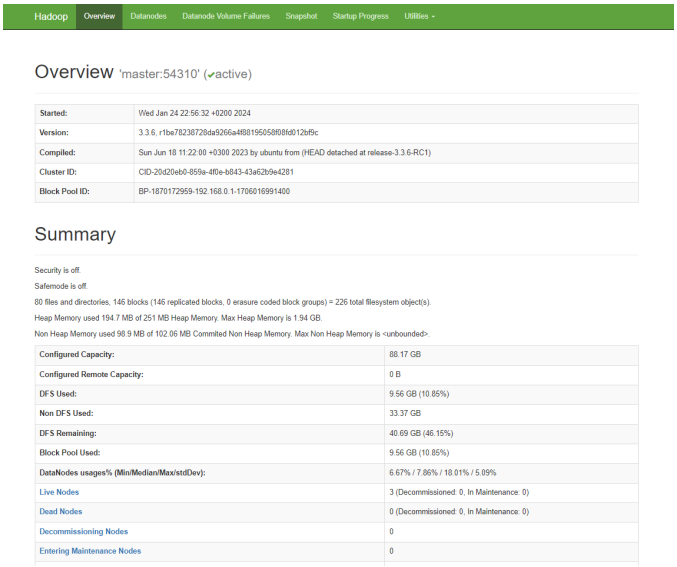

Fig. 1. Screenshot of HDFS Web Interface

Consecutively, to configure the environment for Hadoop YARN, we alter the contents of the file yarn-site.xml appropriately (changes also available on the git repository). We confirm the correct functioning of the system by checking the web interface using our master's public IP and the correct port (8088). There we can see that we have 3 active nodes.

B. Hadoop, Apache Spark, Apache YARN

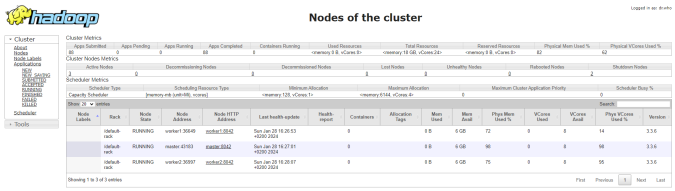

Fig. 2. Screenshot of YARN Web Interface

We installed Apache Spark over Yarn on our newly created cluster of virtual machines. We started by installing the Java version supported by the official repositories of our operating system on all the nodes in the cluster. We then created a special directory to store all executables of the latest versions. We downloaded the compressed files from the official websites of Hadoop and Apache Spark and decompressed them. The executables were moved to the new directory and soft links were created to the main directory. The file /.bashrc was modified to add variables containing the paths to these executables and other useful variables that are included in the git repository linked on the first page.

To configure the environment for the Hadoop Distributed File System (HDFS), we perform further necessary adjustments in the files hadoop-env.sh, core-site.xml, hdfs-site.xml, and create a new file containing the names of our virtual machines. All changes can be found in our git repository. Through a browser we can access the HDFS web interface using our master's public IP and the correct port (9870).

Lastly, to configure the environment for Apache Spark, we create the file spark-defaults.conf where we define the basic properties of our task execution environment and add some important content, available on the git. We also create an HDFS directory to store all log data produced by the execution of our code and launch the Spark history server. We confirm the correct functioning of the system by checking the web interface using our master's public IP and the correct port (18080).

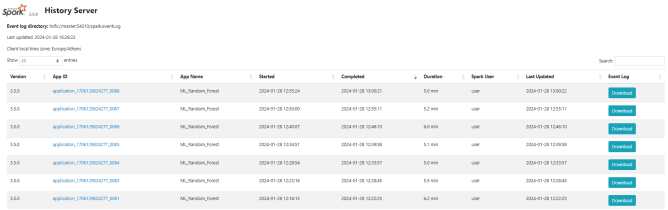

Fig. 3. Screenshot of Apache Spark Web Interface

For a script to be executed in Spark, it has to build and configure a spark session in the beginning.

C. Ray

We installed Ray for general Python applications on every node from the command line using the default option. To use Ray, we run the command ”ray start –head” from the master node and we add the other nodes to the cluster by running the command ”ray start –address=¡address¿” that Ray proposes from the other nodes. In the output of ”ray start –head”, Ray also provides us with a link to its web interface.

In order for a script to be executed with Ray, we also have to add the lines: ”import ray”, ”ray.init()” in the beginning of the script.

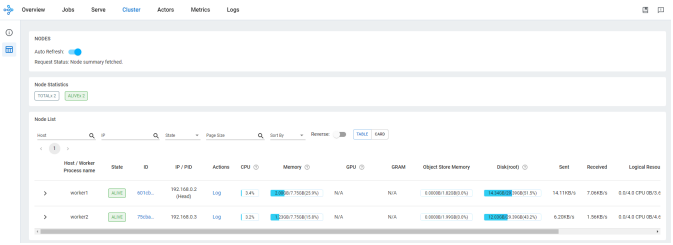

Fig. 4. Screenshot of Ray’s Web Interface

IV. Data generation

A. Machine learning dataset generation

We developed a Python script to create a sizable dataset in CSV format through command-line inputs, specifically allowing the user to define the number of rows, features (columns), and the output file name. The values for each row are randomly generated. The script utilizes the argparse library for handling command-line arguments, pandas for data manipulation, and numpy for generating random data. It employs a function to generate the dataset in chunks, with each chunk written to the specified output file. The goal is to produce a large dataset suitable for various data analysis and machine learning tasks, with the flexibility to control the dataset’s size and structure through the provided command-line parameters. We do not care that the data makes sense, we simply want a dataset that resembles an actual machine learning dataset so that we can simulate the calculations and its manipulation.

For our experiments we chose to create 4 datasets with this script. They have the following characteristics:

1) Dataset A:

- 10 million rows
- 100 features
- size: 5.7 GB

2) Dataset B:

- 18 million rows
- 100 features
- size: 8.8 GB

3) Dataset C:

- 15 million rows
- 120 features
- size: 8.8 GB

4) Dataset D:

- 13 million rows
- 90 features
- size: 5.7 GB

As one can notice, all datasets are of significant size. The datasets B, C were purposefully made the same size in order to evaluate the impact of the number of rows and the number of features. For the same reason, the datasets A, D are also of the same size.

B. Graph generation

To generate graph data to use for our research we created a C++ program, which designs a random graph with n vertices and edges between vertices based on a given probability p.

More specifically, the C++ program takes three command-line arguments: n (number of vertices), p (probability of edge existence between vertices), and the filename to save the generated graph. It uses a simple pseudorandom number generation technique, seeded with the current time, to determine whether an edge should exist between each pair of vertices. The probability p controls the likelihood of an edge between any two vertices. The program iterates through all possible vertex pairs, and for each pair, it generates a random float between 0 and 1. If this random value is less than p, an edge is added between the corresponding vertices, and the edge information is written to the output file. The resulting file represents a graph in an adjacency list format, where each line indicates an edge between two vertices.

This C++ program is a concise and effective tool for generating random graphs with specified properties. We use this program to create 10 random graphs, which are then given as input for the PageRank function.

V. Experiments

A. Scripts used

For both Apache Spark and Ray, we created scripts that execute the algorithms k-means, random forest and pagerank for our dummy data. Below, we explain how each script works.

1) Apache Spark:

- ***K-means clustering script:***The Python script created orchestrates a data processing pipeline using PySpark, an interface for Apache Spark, to perform K-means clustering on large-scale datasets. The script follows a systematic sequence of operations, beginning with the initialization of a Spark session, essential for leveraging Spark’s distributed computing capabilities. Through the ingestion of training and prediction datasets in CSV format, the script seamlessly transforms the raw data into Spark DataFrames. A critical preprocessing step involves the creation of a feature vector using a VectorAssembler, consolidating individual features into a unified structure.

The K-means clustering model is subsequently configured and trained on the prepared training dataset.

As the script transitions to the prediction phase, it adeptly applies the trained clustering model to the new dataset, facilitating the generation of insightful performance metrics. These metrics, indicative of the model's accuracy and efficiency, are meticulously recorded in a structured format for further analysis. The script captures vital statistics encompassing total execution time, data loading time, processing time, memory utilization, and overarching details about the Spark application, including the number of nodes involved. By employing this comprehensive approach, the script not only achieves the primary goal of clustering data but also provides a robust framework for evaluating the model's performance under varying conditions. The recorded metrics, stored in a designated CSV file, serve as a valuable resource for assessing the scalability and efficacy of the K-means clustering algorithm within a Spark environment.

- ***Random Forest script:***This Python script orchestrates the execution of a machine learning pipeline for training and predicting using a Random Forest Regressor in a PySpark environment. Commencing with the establishment of a Spark session, the script ingests training and prediction datasets in CSV format, reading them into Spark DataFrames. Critical preprocessing steps involve the creation of feature vectors using VectorAssembler, combining individual features into a unified structure. The Random Forest Regressor model is defined, configured, and subsequently trained on the prepared training dataset. As the script transitions to the prediction phase, it reads the new dataset and transforms it into the appropriate format using the previously defined VectorAssembler. Following this, the trained Random Forest Regressor model is applied to make predictions on the new dataset. The script efficiently measures and records the total execution time, data loading time, processing time, memory utilization, and other essential statistics. These performance metrics are systematically organized and stored in a designated CSV file, providing a comprehensive overview of the model's effectiveness in terms of prediction accuracy and resource utilization. The script concludes with an assessment of the number of rows, features, and memory size, contributing valuable insights into the scalability and efficiency of the Random Forest Regressor algorithm within a Spark environment.
- ***PageRank script:*** This Scala script executes a Spark-based graph analytics application using the GraphX library. The program's primary purpose is to execute various graph algorithms, such as PageRank, on a large-scale graph loaded from an HDFS file. Currently, only a PageRank function is included, but more can be added. The script incorporates SparkMeasure to collect and analyze performance metrics during the execution.

  The pageRank function loads the graph from the specified HDFS file and runs the PageRank algorithm. It measures the loading time, processing time, number of vertices, and number of edges. The loading and processing steps are retried up to a specified number of attempts to handle potential failures.

  In the main function, the script parses command-line arguments for the algorithm to execute and the graph file path. It creates a Spark session, sets up SparkMeasure for monitoring, and executes the specified graph algorithm. The performance metrics, along with additional metrics extracted from SparkMeasure, are aggregated, and a summary is printed. The results are also written to an HDFS folder for further analysis.

2) Ray:

- ***K-means clustering script:***The Python script that we developed coordinates a machine learning pipeline for K-Means clustering using the Ray framework. Initially, the script initializes the Ray cluster and loads the dataset for model training. It then parallelizes the K-Means model training across the Ray cluster, leveraging distributed computing capabilities. The K-means model is initialized and trained on the dataset using the KMeans implementation from scikit-learn.

  To efficiently handle large datasets, the script divides the workload into multiple tasks using Ray tasks, each responsible for clustering a specific subset of the data. The script measures the time taken for model training and predicts the clusters for each partition of the dataset in parallel. The results, including predicted clusters and memory usage for each partition, are aggregated and recorded.

  The script calculates the overall execution time, memory utilization, and other essential statistics. These metrics are then organized and stored in a CSV file, offering a comprehensive overview of the K-Means clustering algorithm's performance in terms of prediction accuracy and resource consumption. The script concludes with information about the number of rows and features in the dataset, providing insights into the scalability and efficiency of K-Means clustering within a Ray distributed computing environment.
- ***Random Forest script:***The created Python script utilizes Ray for distributed computing in the context of training a Random Forest Regressor model and making predictions on a large dataset. The script employs Ray tasks to parallelize the prediction process across the Ray cluster, enabling efficient processing of substantial amounts of data.

  Initially, the script initializes Ray on all nodes and loads the training dataset. The Random Forest Regressor model is then initialized and trained on the dataset. Following the training phase, the script parallelizes the prediction task using Ray tasks, where each task is responsible for predicting a specific subset of the dataset. The script measures the time taken for model prediction and aggregates the predictions and memory usage results.

The overall execution time, memory utilization, and other essential statistics are calculated and stored in a CSV file. These metrics provide valuable insights into the efficiency and scalability of the Random Forest Regressor model within a distributed computing environment powered by Ray. The script concludes by shutting down the Ray cluster. The recorded information includes the total time for the entire process, memory usage, the number of rows in the dataset, and the number of features considered in the analysis.

- ***PageRank script:*** This Python script executes distributed graph analytics applications using the Ray framework. The program is configured to give the user the ability to choose which algorithm is going to be executed (e.g., PageRank), however, we only implemented one algorithm to select (PageRank). The primary components include loading the graph from HDFS, performing the chosen graph algorithm (PageRank), and recording performance metrics such as execution time, memory usage, and the number of Ray nodes.

  The script utilizes Ray to distribute the workload efficiently across multiple nodes. It starts by reading graph data from HDFS in a distributed manner, measuring the loading time and memory usage. Subsequently, it applies the specified graph algorithm (in this case, PageRank) using the NetworkX library, capturing relevant statistics such as the number of vertices, edges, and algorithm execution time.

  The main function orchestrates these tasks, checks for valid algorithm names, and prints the results along with additional system information like the number of CPU cores and Ray nodes. The gathered performance metrics, including execution time and memory usage, are then written to a CSV file for further analysis.

B. Conduction of the experiments

1) Machine learning experiments: We executed the 4 scripts on Spark and Ray for every dataset. More specifically, for each dataset we ran each script for a medium of 5 times in order to reduce bias and outliers. All results and relevant statistics were saved to be interpreted later on.

2) Graph experiments: We create a bash script to utilize both the graph generation and the PageRank script. This bash script performs the following tasks in a loop for 10 iterations:

1. Generates a random integer between 1001 and 2000.
2. Sets a fixed float value of 0.75.
3. Calls the external C++ program for graph generation with the generated random integer as the number of vertices, the fixed float as the possibility for edge existence, and a filename as the output file to create a graph file.
4. Copies the generated graph file to HDFS.
5. Submits a Spark job using spark-submit to perform PageRank on the graph stored in HDFS.
6. Submits a Ray job using ray job submit to execute the Python PageRank script that performs PageRank on the graph in HDFS using Ray.
7. Removes the graph file from HDFS.

The graphs created and given as input for PageRank have a number of nodes between 1001 and 2000 and approximately 1,000,000-2,000,000 edges. All results are stored by the PageRank scripts to be analyzed later on.

VI. Results from the Machine Learning experiments

After accumulating the results for every experiment we ran, we calculated the mean values for each useful characteristic. All statistics calculated can be seen in the tables:

A. K-means clustering

1) Apache Spark: Information gathered from executing the Apache Spark code:

Dataset	Number of rows (M)	Number of features	Number of training rows	Number of nodes	Total time (sec)	Load data time (sec)	Process data time (sec)
A	10	100	2000	3	133.1	127.6	5.2
B	18	100	2000	3	218.5	214.4	3.8
C	15	120	2000	3	209.8	206.8	2.8
D	13	90	2000	3	181.2	172	8.8

TABLE I
Results for Spark executions

As one can notice from the table, each time the model was trained using a smaller dataset of 2000 rows. For each execution of the script, three nodes participated.

The metric ”Total time” corresponds to the time it took for the script to load the dataset, make predictions and also print relevant information. In other words, it includes all the actions apart from the loading of the training dataset and the actual training of the model. Dataset B has the longest ”Total time”, with 218.5 seconds, followed by Dataset C (209.8 seconds), Dataset D (181.2 seconds) and Dataset A (133.1 seconds).

The metric ”Load data time” corresponds to the time needed to load the dataset. Apparently, it takes up a significant portion of the ”Total time”, suggesting data loading is a non-negligible factor. Again, Dataset B has the longest ”Load data time”, with 214.4 seconds, followed by Dataset C (206.8 seconds), Dataset D (172 seconds) and Dataset A (127.6 seconds).

The metric ”Process data time” refers to the time spent on the actual clustering process of the prediction. Dataset D has the highest process time (8.8 seconds), followed by Dataset A (5.2 seconds), while Datasets B and C have lower process times 3.8 seconds and 2.8 seconds respectively.

From the above observations, we can conclude that the execution times are influenced by both the dataset size and the number of features. Datasets B and C, with larger sizes and more features, naturally took more time to load. On the other hand, the processing time does not seem to be influenced by the size of the dataset. Moreover, for Datasets B, C which have the same size (8.8 GB) we can observe that C, which has more features and fewer rows, takes less time to load and

execute than B and the same is true for Datasets A and D, which also have the same size (5.7 GB). However, the larger Datasets B and C need less processing time than the smaller ones A and D. This might mean that there exists a connection between the number of rows and the number of features to achieve lower execution times, an ideal ratio.

2) Ray: Information gathered from executing the Ray code using 3 nodes:

Dataset	Number of rows (M)	Number of features	Number of training rows	Number of nodes	Total time (sec)
A	10	100	2000	3	829.8
B	18	100	2000	3	7583
C	15	120	2000	3	4327
D	13	90	2000	3	3755

TABLE II
RESULTS FOR RAY EXECUTIONS

For Ray we have information only regarding the total time of the execution, that is without the training, as Ray performs both loading and processing in a parallelized manner, making it impossible for us to accurately calculate these specific metrics.

We can observe from the table that Dataset B has the biggest total time of 7583 seconds, which is approximately 2 hours. It is followed by Dataset C with 4327 seconds, Dataset D with 3755 seconds and Dataset A with 829.8 seconds. As a result, we can conclude that the more rows a dataset has, the longer the execution time is going to be.

We also executed this code using 2 nodes and we got the following results:

Dataset	Number of rows (M)	Number of features	Number of training rows	Number of nodes	Total time (sec)
A	10	100	2000	2	938
B	18	100	2000	2	7441
C	15	120	2000	2	1101
D	13	90	2000	2	3616

TABLE III
RESULTS FOR RAY EXECUTIONS WITH 2 NODES

It is apparent that there exists no improvement from increasing the number of nodes. Unfortunately, this was because our master node was attacked during our experiments which resulted in its resources being constantly monopolized by malware. Consequently, when it was running alongside the two workers, it could not offer much support and speed up the process. On the contrary, because of its involvement, the time to complete each execution actually increased.

3) Comparison: Looking at the execution times of Apache Spark and Ray for all datasets, it is obvious that Spark is much faster than Ray for our implementations. More specifically, Spark is sometimes even more than 10 times faster than Ray. However, the relative differences between the datasets remain unchanged for both Ray and Spark. We have to underline though that the results gathered for Ray are not accurate, as we did not accomplish execution using 3 nodes.

B. Random Forest

1) Apache Spark: Information gathered from executing the Apache Spark code:

Dataset	Number of rows (M)	Number of features	Number of training rows	Number of nodes	Total time (sec)	Load data time (sec)	Process data time (sec)
A	10	100	2000	3	125.4	121.9	3.3
B	18	100	2000	3	214.9	211.4	3.3
C	15	120	2000	3	214.9	209.3	5.2
D	13	90	2000	3	143.3	138.7	4.1

TABLE IV
RESULTS FOR SPARK EXECUTIONS

As one can notice from the table, each time the model was trained using a smaller dataset of 2000 rows. For each execution of the script, three nodes participated.

Datasets B and C have the longest ”Total time”, with 214.9 seconds, followed by Dataset D (143.3 seconds) and Dataset A (125.4 seconds).

Dataset B has the longest ”Load data time”, with 211.4 seconds, followed by Dataset C (209.3 seconds), Dataset D (138.7 seconds) and Dataset A (121.9 seconds).

Dataset C has the highest process time (5.2 seconds), followed by Dataset D with 4.1 seconds, while Datasets A and B have lower process times at 3.3 seconds.

Our findings here are similar to our findings for the K-means clustering algorithm. Again, the execution times are influenced by both the dataset size and the number of features. Datasets B and C, with larger sizes and more features, naturally took more time to load.

For Datasets B and C, which have the same size but different numbers of rows and features, we can observe that, although their execution times are identical, Dataset C spends more time being processed than Dataset B, or in other words, Dataset B needs more time to load than C. The longer loading time for B can be attributed to B having more rows and the longer processing time for C can be attributed to it having more features. For Datasets A and D however, even though they share the same relationship as do B and C, it is D that has both the longest loading and longest processing time. This fact leads us to assume that there is a correlation between the number of rows and the number of features, which also affects the processing time. In other words, there exists a perfect ratio for low execution time.

In comparison to the K-means clustering algorithm, it appears that apart from the slight changes in the processing times, the overall execution times are similar.

2) Ray: Information gathered from executing the Ray code using 3 nodes:

Dataset	Number of rows (M)	Number of features	Number of training rows	Number of nodes	Total time (sec)
A	10	100	2000	3	1635
B	18	100	2000	3	7437
C	15	120	2000	3	4296
D	13	90	2000	3	3616

TABLE V
RESULTS FOR RAY EXECUTIONS WITH 3 NODES

On this table again we only have information regarding the total execution time, without the training stage. Similarly to our findings for the K-means algorithm, Dataset B has the biggest total time of 7437 seconds, which is approximately 2 hours. It is followed by Dataset C with 4296 seconds, Dataset D with 3616 seconds and Dataset A with 1635 seconds. Again we can conclude that the more rows a dataset has, the longer the execution time is going to be. In addition, we notice a significant increase in the average execution time for Dataset A compared to the execution time for the K-means algorithm. The remaining datasets have slightly lower execution times for the Random Forest algorithm compared to the execution times for the K-means algorithm.

We also executed this code using 2 nodes and we got the following results:

Dataset	Number of rows (M)	Number of features	Number of training rows	Number of nodes	Total time (sec)
A	10	100	2000	2	996
B	18	100	2000	2	7142
C	15	120	2000	2	1285
D	13	90	2000	2	3616

TABLE VI
RESULTS FOR RAY EXECUTIONS WITH 2 NODES

Again, increasing the number of nodes did not improve the execution time for the reasons mentioned previously.

3) Comparison: Once again, looking at the execution times of Apache Spark and Ray for all datasets, it is obvious that Spark is much faster than Ray for our implementations. Just like before, Spark is many times faster than Ray, but the relative differences between the datasets are equivalent for both Ray and Spark. However, we have to keep in mind that we did not manage to get accurate results from Ray by executing the programs using 3 nodes.

VII. RESULTS FROM THE GRAPH EXPERIMENTS

After accumulating the results for every experiment we ran, we calculated the mean values for each useful characteristic. All statistics calculated can be seen in the tables:

A. Apache Spark

Information gathered from executing the Apache Spark code:

Graph	Number of vertices	Number of edges	Total time (sec)	Proc. time (sec)	Load time (sec)	Exec. run time (sec)	Exec. CPU time (sec)	Load memory (GB)	Number of nodes
1	1090	891665	26.1	17.4	9.2	19.5	14	0.08	3
2	1099	906415	18.9	14	5.4	16	13.1	0.06	3
3	1074	980763	23.4	16.5	8.4	18.1	13.3	0.02	3
4	1184	1191201	24	17.5	8.1	18.8	14.3	0.04	3
5	1225	1125223	26.4	18.5	9	22	15.7	0.06	3
6	1332	1330189	18.4	14.5	5.3	17.5	14.4	0.08	3
7	1343	1352570	23.5	17	7.4	19.9	14.6	0.06	3
8	1363	1392672	24.3	16.6	8.6	20.9	15.3	0.07	3
9	1479	1858117	25.1	17.6	8.2	22.2	16.8	0.06	3
10	1614	1953913	26.3	18.5	9.1	23.7	17.7	0.07	3
11	1631	1993868	25.8	18.2	8.7	22.5	17.5	0.08	3
12	1737	2262802	26.5	18.8	8.9	24.4	18.7	0.05	3

TABLE VII
RESULTS FOR SPARK EXECUTIONS

We have ordered the graph datasets in the table based on their size. Because we represented the graphs as text files where each line represents an edge, the more edges a graph has, the bigger its size in GB.

The 'Total time' refers to the time needed for the whole PageRank program to be executed and the 'Load time' refers to the time needed for the graph to be loaded. 'Process time' refers to the time needed for the PageRank function to execute with the graph as input. 'Executor run time' is the time each spark executor works for and 'Executor CPU time' is the amount of time spent in the CPU of each executor. 'Load memory' is the memory needed to load the graph.

In general, we can observe that all time metrics increase as the size of the graphs increases. However, there is also a certain degree of randomness, as for example, the smallest graph has approximately the same 'Total time' as the largest. The memory needed to load is small and roughly the same for all graphs.

B. Ray

Information gathered from executing the Ray code:

Graph	Number of vertices	Number of edges	Total time (sec)	Load time (sec)	Processing memory (GB)	Load memory (GB)	Number of nodes
1	1090	891665	3.8	1	6.5	6.2	3
2	1099	906415	3.5	1	6.5	6.2	3
3	1074	980763	4.5	1.1	6.6	6.2	3
4	1184	1191201	5.2	1.3	6.6	6.2	3
5	1225	1125223	5.1	1.3	6.6	6.2	3
6	1332	1330189	6	1.4	6.7	6.2	3
7	1343	1352570	6	1.5	6.6	6.2	3
8	1363	1392672	6.5	1.5	6.5	6	3
9	1479	1858117	9	2	6.9	6.3	3
10	1614	1953913	9.5	2.1	6.9	6.3	3
11	1631	1993868	9.6	2.1	6.9	6.3	3
12	1737	2262802	11.5	2.4	6.9	6.2	3

TABLE VIII
RESULTS FOR RAY EXECUTIONS

We can observe that the larger the graph the longer the total time for execution is. In some cases this rule is broken, however, in these cases, the size difference between the graphs is very small and so is the execution time difference. The load time follows the same pattern, even more religiously. The load memory is nearly identical for all graphs and the processing memory is quite similar for all too, with subtle increases in memory needs for bigger graphs.

C. Comparison

It is obvious from our results that Ray is much faster than Spark for our implementations. More specifically, it is easy to understand from the tables that Ray is 4-5 times faster than Spark. However, Ray needs a lot more memory for the graphs to load and be processed in comparison with Spark.

VIII. Conclusion

In conclusion, our research delved into a comprehensive comparison between Apache Spark and Ray in the context of big data analysis and machine learning. By assessing various metrics, including execution time and memory usage, we conducted copious tests on the k-means algorithm and random forest for voluminous datasets. Additionally, we explored the performance of both frameworks on the PageRank algorithm for graph analysis.

The findings of our study revealed a nuanced picture, showcasing distinctive strengths of each framework in specific scenarios. Apache Spark demonstrated superior speed in the execution of the k-means algorithm and random forest, making it a compelling choice for tasks demanding swift processing. On the other hand, Ray exhibited remarkable efficiency in the context of the PageRank algorithm, outpacing Spark significantly in terms of speed. Despite this, it is noteworthy that Ray tended to require more memory for optimal performance.

In essence, the choice between Apache Spark and Ray should be made judiciously, considering the specific requirements of the task at hand. Depending on the nature of the workload, users may prioritize Spark for its efficiency in k-means and random forest, or opt for Ray when dealing with graph analytics tasks such as PageRank. Our research contributes valuable insights to the ongoing discourse on big data frameworks, offering a nuanced perspective to guide practitioners in selecting the most suitable solution for their distinct use cases.

IX. Code Availability

The code and scripts used in these experiments are open-source and available on GitHub at [https://github.com/Tsoktsok/Information-Systems-NTUA]. Researchers and practitioners interested in replicating or extending our work can access the codebase for further insights and exploration. We encourage the scientific community to leverage and build upon our findings for advancing research in distributed computing and machine learning.

References

[1] Ray Project, Ray Documentation.

[2] Apache Spark Contributors, Apache Spark Documentation.

[3] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., Stoica, I. (2012). Apache Spark: A Unified Analytics Engine for Big Data Processing. In Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation.

[4] Nishihara, R., Moritz, P., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I., Stoica, I. (2018). Ray: A Distributed Framework for Emerging AI Applications. In Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation

[5] Kern, H., Rabl, T., Markl, V. (2016). Comparing Apache Spark and Apache Flink for Scalable Data Processing. In Proceedings of the European Conference on Computer Systems

[6] Khayyat, Z., Kamdar, M. R., Gandhi, R., Gunopulos, D. (2017). A Comparative Study of Spark, Flink, and Kafka for Large-Scale Stream Processing. In 2017 IEEE International Conference on Data Mining.

[7] Xin, R. S., Lian, C., Bradley, J. K., Meng, X., Xie, T., Borkar, V., Franklin, M. J., Ghodsi, A., Shenker, S., Stoica, I., Zaharia, M. (2013). Evaluating Apache Spark. In Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation.