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

Νικόλαος Κάσσαρης
School of Electrical and Computer Engineering
National Technical University of Athens
Athens, Greece
el19188@mail.ntua.gr
*project team:**28***

Κωνσταντίνος Βούγιας
School of Electrical and Computer Engineering
National Technical University of Athens
Athens, Greece
el19144@mail.ntua.gr
*project team:**28***

Abstract*—As the need to process large volumes of data continues to grow, contemporary big data processing frameworks play an increasingly crucial role. This project centers on comparing two specialized systems, Ray and Apache Spark, in the context of Extract, Transform, Load (ETL) tasks and both basic and advanced ML/AI workloads within big data projects. For our experiments, we generate and identify appropriate datasets of varying sizes that are stored in a Hadoop Distributed File System (HDFS) across several machine nodes. Our objective in this study is to evaluate the relative strengths and weaknesses of each system in big data operations, monitoring their efficiency, scalability, and robustness across different workload sizes and resource availability. Our findings indicate that both systems exhibit remarkable capabilities in complementary areas of big data workloads, establishing them as formidable players in this domain; Apache Spark excels at managing ETL operations on large datasets, while Ray transforms Distributed Machine Learning by integrating Ray Datasets with cutting-edge high-performance ML frameworks.

Keywords*—Ray, Apache Spark, Distributed Computing, HDFS, ETL Pipelines.

I. Introduction

In today's digital era, most people engage with the internet daily for various purposes, including social media, online shopping, entertainment, financial transactions, work, and research. The digital world has become an integral part of modern life, leading to an enormous surge in data generation from diverse sources. This rapid data expansion has created an urgent need for efficient methods to store, manage, and analyze vast amounts of information within acceptable timeframes while utilizing sustainable, often distributed, computing resources.

The combination of these extensive datasets and advancements in computational power has been a driving force behind the rapid progress of machine learning in recent years. Continuous research has resulted in increasingly sophisticated algorithms and models with remarkable capabilities to recognize patterns and make accurate predictions. Consequently, machine learning is now widely applied across various domains, solving complex problems and enabling groundbreaking technological innovations.

As a result, a significant portion of scientific research and technological development today revolves around machine learning and big data. To support these advancements numerous frameworks have been designed to streamline and simplify data handling and analysis. Python, being one of the leading programming languages in data science, offers a diverse range of frameworks tailored for these tasks. However, with each framework possessing distinct strengths and unique features, selecting the most suitable one can be a challenging decision.

To tackle this challenge, this paper presents a comparative analysis of two Python-based scaling frameworks: Ray and Apache Spark. In a distributed computing environment, we assess their performance through a series of experiments, utilizing various data types and sizes across different numbers of nodes. Our evaluation includes standard data-management operations (ETL) and well-known machine learning and graph processing tasks, such as Clustering and PageRank and Triangle Counting.

It is important to note that, while we have made a deliberate effort to ensure our experiments are as realistic as possible and yield meaningful insights, our primary objective is not to measure the accuracy of the algorithms but to analyze the scalability and efficiency of Ray and Apache Spark. The core aim of this research is to highlight the strengths, limitations, and key differences between these frameworks, offering valuable insights into their optimal use cases in real-world applications.

II. Code Availability

All the scripts we used to run the experiments of this paper can be found in our GitHub repository [1], available at https://github.com/NtinosVg/NTUA-BigData-Spark-Ray. This Github repository contains all the essential code for replicating our experiments comparing Ray and Apache Spark frameworks, as well as instructions on how to proceed with the Setup and Installation.

The three directories are *Data*, *Scripts* and *Documents*. The *Scripts* directory consisting of subdirectories, named according to the job they are relevant to, are dedicated to the necessary python scripts used, as well as a READ-ME file that explains the necessary commands to run them. The *Data* directory does not contain the actual data sets used and loaded in our Database, but instructions on how to download, install them, or generate them, accompanied by the necessary scripts. The *Documents* directory includes the project report.

III. Infrastructure and Software Overview

A. Okeanos-Knossos [2]

Okeanos-Knossos is a cloud platform provided by GRNET for the Greek Research and Academic Community, offering production-grade IaaS with virtual infrastructures. It is available to Greek academic users at no cost. Okeanos-Knossos allows users to deploy custom virtual machines and networks, manage storage, and create varied environments and network configurations without the need for physical hardware. The service offers a robust, flexible, and easy-to-use environment, ideal for experimenting with various technologies and distributed systems.

B. Apache Hadoop [3], [4]

HHadoop is a framework designed for the scalable and distributed processing of large datasets. It provides a highly available service across a cluster of machines while also detecting and managing failures. In this study, we utilized a component of Apache Hadoop known as the Hadoop Distributed File System (HDFS). HDFS is a distributed file system with a master/slave structure that ensures reliable storage and offers high-throughput data access, making it ideal for applications dealing with large datasets and distributed clusters.

C. Python

Python is one of the most prominent languages in data science and machine learning applications. It is versatile and easy to use, featuring a low learning curve and, most importantly, extensive library support. Python provides a big variety of powerful libraries and frameworks for data analysis and manipulation, machine and deep learning operations, big data processing, data visualization, NLP, automation and many other tasks. The main focus of our paper is to compare two of these Python frameworks, Ray and Apache Spark, but within the scope of our research we used other Python libraries as well, some of which are scikit-learn, numpy, pandas and pyarrow.

D. Ray [5]

Ray is an open-source unified framework designed to scale AI and Python applications, like machine learning. It minimizes the complexity of managing distributed systems, as it provides a compute layer for parallel processing across multiple cores and nodes. Moreover, Ray encapsulates built-in libraries for data processing, model training, hyperparameter tuning and reinforcement learning, that facilitates efficient scaling of machine learning models. With its user-friendly API and the automated handling of key processes, such as scheduling, it simplifies scaling across large clusters. Ray also offers many advantages for working efficiently with datasets, for instance, the Ray Datasets component provides high-performance data processing API for large-scale datasets.

E. Apache Spark [6]

Apache Spark is an open-source framework widely recognized for its high-performance distributed data processing and analytics capabilities across large datasets. It supports both batch and real-time processing using an optimized DAG (Directed Acyclic Graph) execution engine. One of Spark's core components is the Resilient Distributed Dataset (RDD), a fault-tolerant, parallel data structure that enables efficient in-memory computations. Additionally, Spark provides a rich ecosystem of libraries for various data-intensive tasks, including Spark SQL for structured data processing, MLlib for scalable machine learning, GraphX for graph processing, and Spark Streaming for real-time analytics. Furthermore, Spark's distributed computing model allows seamless execution across clusters, optimizing performance and scalability through efficient resource management.

IV. Installation and Setup

A. Virtual Machines

For the purposes of this project we setup a distributed cluster that consists of 5 Virtual Machines provisioned by Okeanos-knossos each with:

- Ubuntu Server 24.04 LTS
- 4 CPUS
- 8 GB RAM
- 30 GB Disk size

B. Network Configuration

All of our VMS are connected to a private network and the master node is assigned a public IP address. The web services of our cluster are available through this public IPv4.

C. Apache Spark and Hadoop Configuration

We followed the *Hadoop Adv DB Guide* for the Hadoop and Spark installation. The guide focuses on installing and setting up the environment for Hadoop and Spark. To start HDFS and YARN setup we use the commands 'start- dfs.sh' and 'start-yarn.sh' on the master node. The Spark history server was also set up to access via a web app the history of the completed tasks. We start the history server using the command '$SPARK_HOME/sbin/start-history-server.sh'.

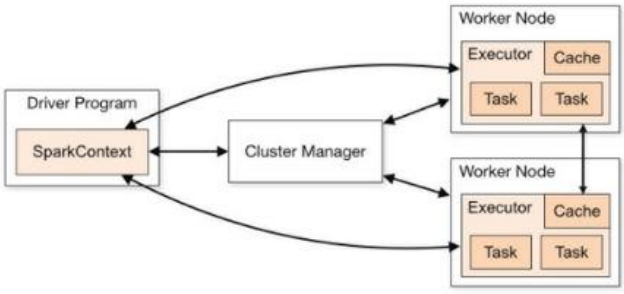

Fig 1: Spark Cluster Overview

D. Ray Setup

We set up a python virtual environment on all our nodes and install the Ray Python package with 'pip install ray' along with all the packages that are found in the requirements.txt file located in the *Documents* directory of our GitHub Repo. We configure a cluster consisting of three nodes. The head node will run some additional control processes (driver process, global control store, autoscaler) [7].

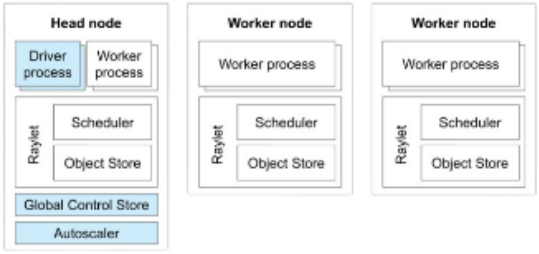

Fig 2: Ray Cluster Overview

We start the Ray process head node, and thus initiate the cluster (with dashboard) by running the following command:

ray start --head --node-ip-address=192.168.1.1

--port=6379 --dashboard-host=0.0.0.0

With these flags we limit RAM usage and enable disk spilling:
--object-store-memory=2147483648
--system-config='{"automatic_object_spilling_enabled": true,
"object_spilling_threshold": 0.8}'

We connect the workers to the cluster with the command:

ray start --address='192.168.1.1:6379'

V. Datasets

For our purposes, we generated synthetic mixed data that contain both categorical features as well as numeric, in a csv format using the *make_classification* function from the *sklearn.datasets* library. We loaded the csv files in our Hadoop Distributed File System and run the *hdfs balancer* command to distribute the data equally among our datanodes. We use these csv files of different sizes (2,4 and 8 GBs) to run our ETL and ML scripts. Table I gives an overview of these datasets and Table III shows the schema of their contents.

We also use graph data in the form of a list of edges to run several graph operations. Using a custom script we generated artificial data and also sourced real graphs through the KONECT project [7] and the Stanford SNAP project [8]. Our data are in tsv file format and here is a brief summary of each graph:

- Scale-Free graphs generated using the Barabasi-Albert model. This type of social network simulates a topology whose degree distribution follows a power law, at least asymptotically. These graphs were generated with NetworkX for number of nodes equal to 500k, 1m, 2m and with an average degree of 5, so each graph has 5 times more edges than vertices.
- Small-World graphs generated using the Wattz-Strogatz model. This type of social network simulates a topology very commonly found in real-world networks. These graphs were generated with NetworkX for number of nodes equal to 50k, 100k, 200k and with an initial Regular Graph degree of 100, and a p value of 0.1. So each graph has 100 times more edges than vertices.
- Lakes: This is the directed road network from the 9th DIMACS Implementation Challenge, for the area "Great Lakes". 2,758,119 nodes and 6,794,808 edges
- Higgs: This is a directed follower social network from Twitter, in the context of the announcement of the discovery of a particle with the features of Higgs boson. 456,626 nodes and 14,855,842 edges.
- Wikipedia links (fr): This network consists of the wikilinks of Wikipedia in the Spanish language (es). Nodes are Wikipedia articles and directed edges are wikilinks, i.e., hyperlinks within one wiki. 3,333,397 nodes and 123,709,902 edges.
- Wikipedia links (en): This network consists of the wikilinks of the Wikipedia in the English language (uk). Nodes are Wikipedia articles, and directed edges are wikilinks, i.e., hyperlinks within one wiki. 13,593,032 nodes and 437,217,424 edges.
- Orkut: This is the bipartite network of Orkut users and their group memberships. 11,514,053 nodes and 327,037,487 edges.

TABLE I: Datasets used for Benchmarking

Rows	#of CSV files	Total Size
25 million	1	2.2 GB
50 million	1	4.4 GB
100 million	1	8.79 GB

TABLE II: Graphs used for Benchmarking

Graph	nodes (n)	edges (m)
SW-*x*K	$x \cdot 1000$	$100 \cdot n$
SF-*x*K	$x \cdot 1000$	$5 \cdot n$
lakes	2,758,119	6,794,808
higgs	456,626	14,855,842
wiki-fr	3,333,397	123,709,902
wiki-en	13,593,032	437,217,424
Orkut	11,514,053	327,037,487

TABLE III: Dataset Schema

f 1	f 2	f 3	f 4	cat 1	cat 2	word	label
float64	float64	float64	float64	int	int	string	(0,1)

VI. Experiments

A. Introduction

In this section, we delve into the theoretical foundations and implementation specifics of our experimental code. Our primary goal is to shed light on the methods and algorithms employed, as well as the various programming choices made during the experiment's development in order to seamlessly run the necessary experiments. For further questions or code details, one must refer to our GitHub repository.

B. Graph Operations

PageRank: PageRank, the algorithm behind Google's

search rankings, assigns each web page a numerical score reflecting its importance based on both the quantity and quality of its incoming links. This experiment not only allows as to compare the distributed execution of Ray and Apache Spark but also provides insight of writing distributed code. We benchmarked the graph datasets using the *graphframes* library in PySpark, which offers an API for GraphX algorithms and naively in Ray using Dataset Transformations.

***Triangle Counting**:* Counting the number of triangles in a graph is a fundamental task in graph theory and network analysis, as triangles—three vertices that are all connected—offer valuable insights into a network's connectivity and structure. In our experiments, we compared two different systems on this problem. For Spark we leveraged the GraphX library, which includes a built-in triangle counting algorithm. In contrast, our ray approach was designed to play to its strengths in parallelizing Python code. We began using the NetworkX library to construct a graph object which we then stored in Ray's object store via ray.put(), ensuring that all worker nodes had access to the same graph. Subsequently, we distributed the workload by employing Ray tasks to assign each worker a subset of the graph's vertices for counting. Again the Network X library provides the triangle counting functionality, processing the list of nodes assigned to each worker. As a result the algorithm runs concurrently on different segments of the graph, enabling multiple machines to work independently on the same overall task.

C. Extract, Transform, Load Operations

Next we assessed the performance of each system using a range of ETL operations which server as the fundamental tasks involving transformations and manipulations of Spark DataFrames/RDDs and Ray Datasets. For each ETL task we record metrics such as Runtime, CPU execution time and peak memory usage. The evaluated tasks include:

- We load large amounts of data from the HDFS. To materialise the Spark Dataframe and the Ray Dataset, we call an action like count() in Spark or materialize() in Ray to trigger the lazy execution of the load command.
- We perform *Group By* and *Aggregation* operations. In both Spark and Ray we will group the dataset by the values of cat_2 column and then find the sum of the f_3 feature across those groups.
- We sort our data based on a specific column. We randomly selected the column f_2 for our experiments.
- We transform the dataset. In our scripts we added a new feature that is the result of the following: $f_1^2 + f_2^{2\ 2}$. Then, we filter out the rows for which the length of the value of the column *word* is larger than the value of the new feature.

We performed these ETL tasks on our synthetic dataset we generated which includes 3 CSV files of different sizes, stored in our HDFS cluster. We access these files through the HDFS and convert to Spark Dataframe and Ray Datasets accordingly to evaluate the systems. The datasets are big enough to not be able to fit in the main memory of a single machines. To achieve accurate results and extrapolate valuable conclusion we also examined the performance of the two setups using different amount of workers/executors.

D. Kmeans Clustering

Clustering is a key technique in machine learning, used to group data points in distinct clusters based on their similarities. One of the most widely used clustering algorithms is k-means, which partitions the dataset into k cluster. The algorithm iteratively assigns data points to the nearest cluster centroid, where the centroid is the mean of all data points within the cluster, and then recalculates the centroid. The process continues until the centroids stabilize, either when their values no longer change significantly between iterations or when a predefined number of iterations is reached. In our experiment we use the generated csv files as the input. For Ray we employed the pyarrow library to read data from HDFS in predefined sized batches. These batches were transformed into the right format and processed in parallel within our cluster. Using transformations on our ray dataset we ran the kmeans algorithm to identify the centroids. We also calculated the runtime of the operation to compare it with the Apache Spark script. Apache Spark comes with the MLLib library that provides many functions for distributed Machine Learning. Ray on the other hand integrates Ray Datasets with many popular state-of-the-art Machine Learning frameworks (PyTorch, Tensorflow, XGBoost), leveraging their capabilities to enable distributed training of powerful ML models on large amounts of data

E. Linear Regression prediction

Linear regression is a fundamental technique in machine learning used for predicting a continuous target variable based on input features. It models the relationship between the dependent and independent variables by fitting a linear equation to the observed data. The model learns the optimal coefficients by minimizing the error, typically using methods such as Ordinary Least Squares (OLS) or Gradient Descent. In our experiment, we used generated CSV files as input data. For Ray, we employed the PyArrow library to read data from HDFS in predefined-sized batches. These batches were transformed into the appropriate format and processed in parallel within our cluster. We trained a SGD Regressor linear model from the *sklearn* library and assessed its R^2 and RMSE scores to compare it to Apache Spark. Together, these two metrics give a **complete picture**: one measures goodness-of-fit, the other measures predictive accuracy. We also measured the runtime of the operation. As Apache Spark comes with the MLlib library, we trained our model and ran the appropriate model transform and fit functions available from the MLlib API.

VII. Benchmarks and Results

This part of the study presents the experiment results and provides a detailed analysis comparing Ray and Apache Spark across multiple performance metrics. We showcase the benchmark results for the tasks described in the previous section, executed on the systems. These tasks were run with varying numbers of workers in the cluster (1, 3 and 5) and different dataset sizes (as shown in Table I and Table II). The key metrics analyzed include Runtime, and Peak Heap Memory. To collect these metrics from tasks running across the cluster, we utilized *SparkMeasure* for spark and *stats()* function for Ray Datasets. For some tasks we also used the *time* library to

estimate the Runtime of the operations. To monitor the activity of the worker nodes we used the Ray Dashboard available at http://83.212.75.179:8265/ and the YARN dashboard available at http://83.212.75.179:8088/ (for ray and spark respectively).

A. ETL Operations

Our experiment was conducted using as input the 3 generated datasets of sizes 2.2,4.4 and 8.8 GB and utilizing 1,3 and 5 worker nodes. For each operation we analyze the results of our testing:

Load-Operation:

Both frameworks perform adequately in this task on each dataset size, with Spark executing faster as shown in table IV. Also, Spark seems to also be more efficient in this task when we consider memory usage.

TABLE IV:Load Runtime and Memory Usage

		Ray		Spark	
Nodes	Dataset (GB)	**Runtime (s)**	**Peak Heap Memory**	**Runtime (s)**	**Peak Heap Memory**
5	2.2	69.8	2331 MB	25	429 MB
3	2.2	61.42	2271 MB	43	786 MB
1	2.2	35.14	2300 MB	50	404 MB
5	4.4	119.8	3023 MB	34	552 MB
3	4..4	78.4	3112 MB	52	566 MB
1	4.4	66.01	2892 MB	78	483 MB
5	8.8	166.6	3186 MB	52	660 MB
3	8.8	135.5	3013 MB	66	524 MB
1	8.8	127.3	3023 MB	174	562 MB

Sort-Operation.

In this task we see Spark vastly outperform Ray, being twice as fast with the same number of nodes and the same dataset.

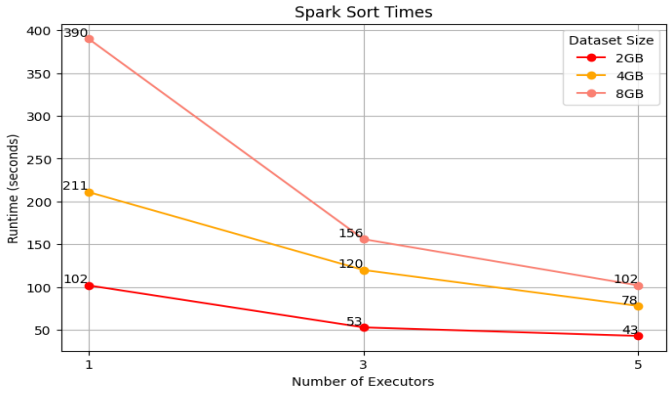

Fig 3: Spark Sort Operation Runtime

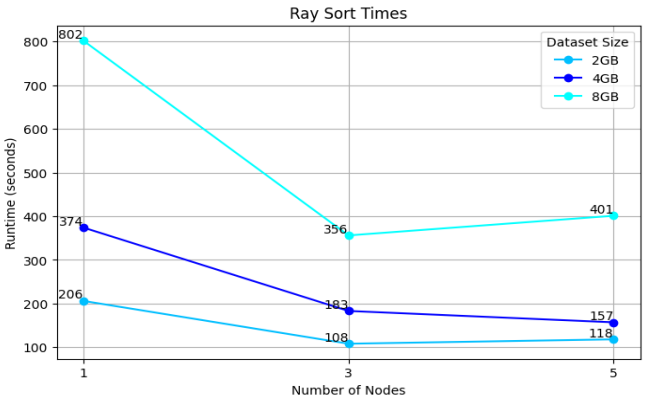

Fig 4: Ray Sort Operation Runtime

We observed a lot of disk spillage during our Ray experiment, especially with our largest dataset of 8GB, which possible hindered the performance of the framework. In order for our Ray experiments to not end prematurely due to OOM (Out of Memory) errors we had to enabled Push-based shuffle. Ray's push-based shuffle operation is an experimental feature, that allows the data to be shuffled from all of the input partitions to all of the output partitions and may improve performance when performing these operations. Ray's. documentation [10] also states that shuffling can be challenging to scale to large data sizes, so it is not out of the ordinary that we noticed a lot of these errors popping up.

Aggregate-Operation Spark appears to perform way better on aggregate tasks. This is reflected on these figures where we see that Spark is up to 10 times faster than Ray:

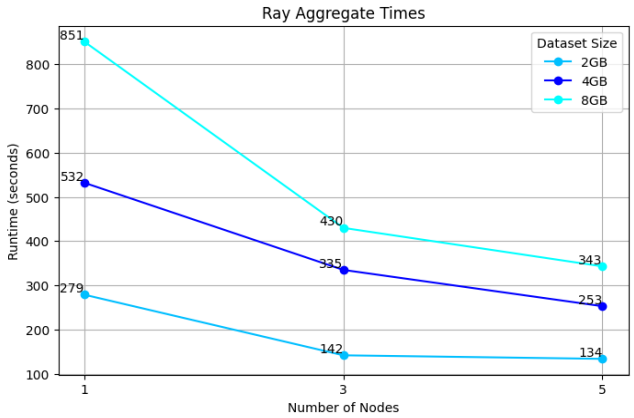

Fig 5: Ray Aggregate Operation Runtime

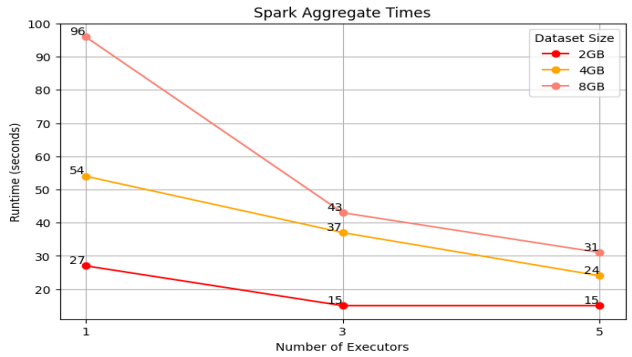

Fig 6: Ray Aggregate Operation Runtime

Transform-Operation: Same with the load and aggregation tasks, Spark performs better than Ray in our transform script, as shown in these figures:

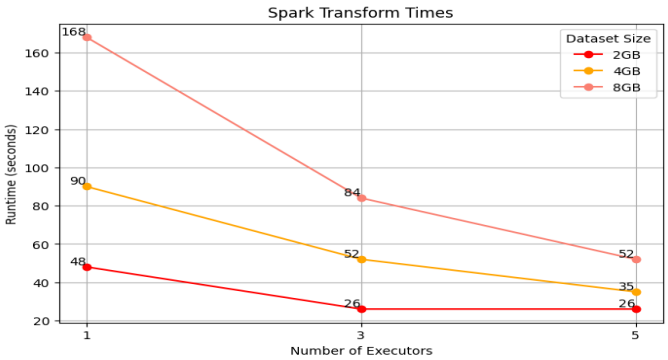

Fig 7: Ray Transform Operation Runtime

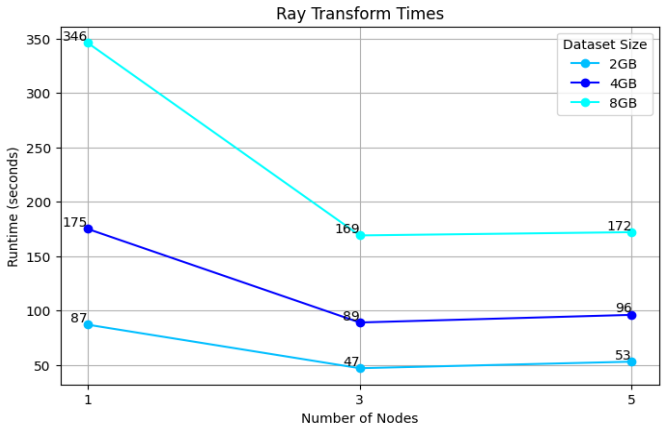

Fig 8: Ray Transform Operation Runtime

B. Graph Operations

Triangle Counting: We ran the experiments on 3 generateed graphs using 5 nodes and the runtime results for this experiment are presented in the figure below. It is evident that the two frameworks have very similar performance in this operation with minimal differences when we scale the dataset to a higher graph node count. The execution runtime between our two scripts does not seem to differ that much even though we use GraphX in Spark and NetworkX in Ray.

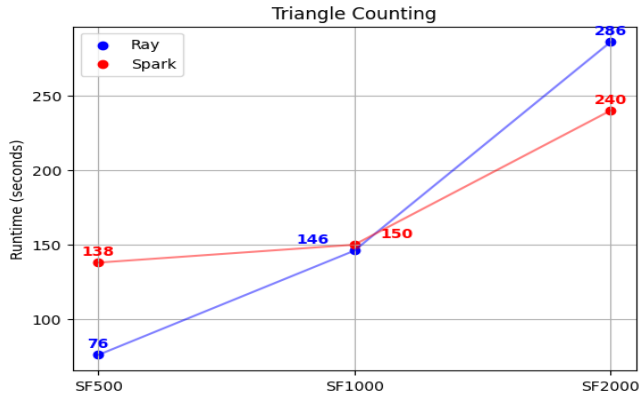

Fig 9: Ray and Spark Triangle Counting Runtime

PageRank: Again the runtime results for this experiment are presented in the figure below. Similarly to triangle counting, there does not appear to be a big difference between the performance of the two frameworks in this operation.

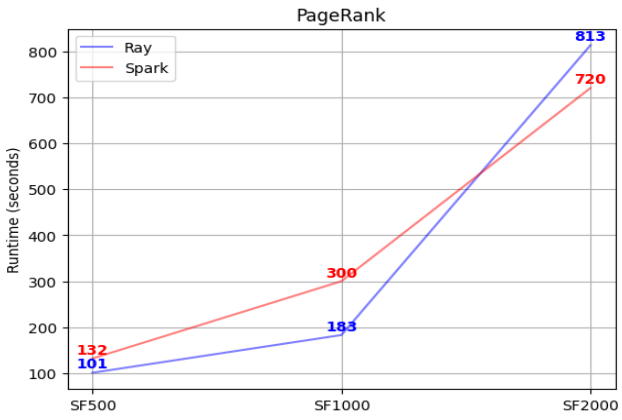

Fig 10: Ray and Spark PageRank Runtime

C. ML Operations

K-Means clustering: We will now show the results of our tests regarding our K-means clustering operation using the figures below. Ray appears to be substantially faster in this operation compared to Spark's MLlib script. Spark seems to scale well in this operation with 5 nodes whereas Ray does not seem to utilize the increased amount of worker nodes for this operation.

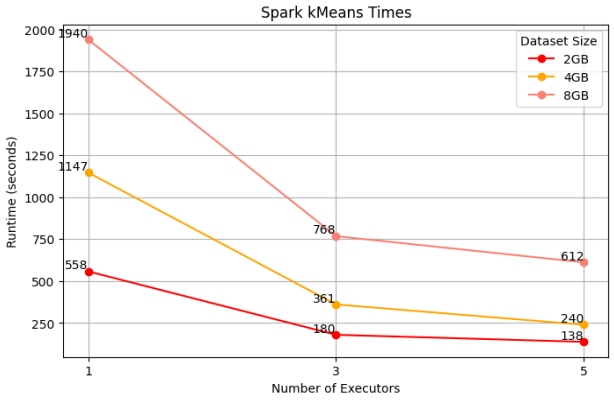

Fig 11: Spark K-Means Operation Runtime

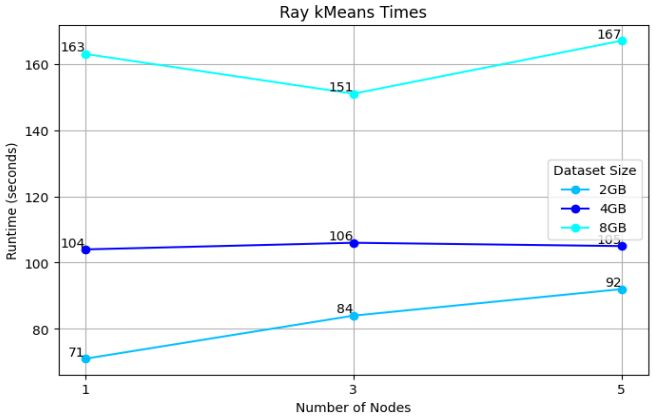

Fig 12: Ray kMeans Operation Runtime

When we consider the peak heap memory usage now the results are inversed to the ETL tasks we performed before. Now Spark appears to significantly surpass Ray in this metric as is shown in TABLE V.

TABLE V:

Workers	Dataset (GB)	Ray Peak Heap Meamory	Spark Peak Heap Meamory
5	2.2	237 MB	2.0 GB
3	2.2	1011 MB	2.1 GB
1	2.2	1147 MB	2.8 GB
5	4.4	257 MB	2.7 GB
3	4.4	322 MB	2.8 GB
1	4.4	1154 MB	2.8 GB
5	8.4	333 MB	2.7 GB
3	8.4	399 MB	2.9 GB
1	8.4	422 MB	3.0 GB

Linear Regression prediction :We will now show the results of our tests of our Linear Regression prediction operation using the figures below. Again Ray does not appear to scale well in this operation with multiple worker nodes unlike Spark which shows big improvements when looking at the different execution runtimes when having 3 or 5 nodes in our cluster compared to 1 node. Ray beats Spark in this

operation with the 1 or 3 nodes but Spark ultimately surpasses Ray in runtime execution with 5 worker nodes.

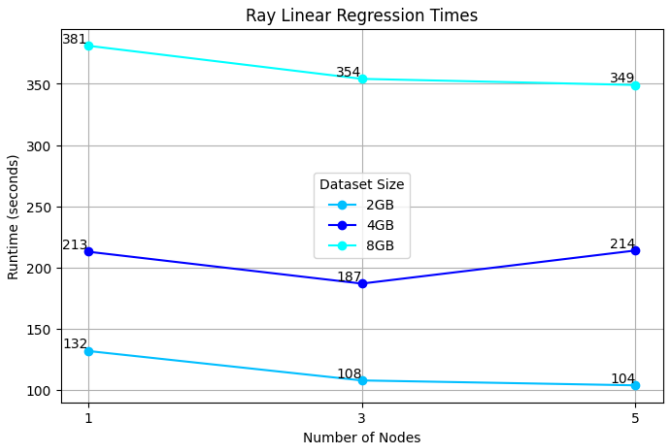

Fig 13: Ray Linear Regression Operation Runtime

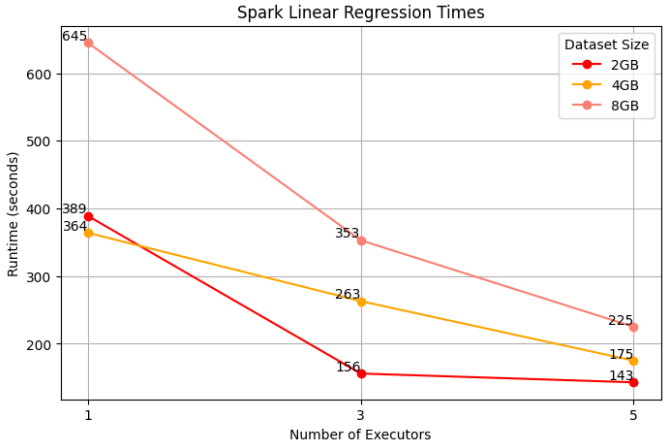

Fig 14: Ray Linear Regression Operation Runtime

When analyzing the R^2 and *Root Mean Squared Error* metrics Spark does not have any significant changes in these metrics with different datasets or worker nodes. However Spark seems to offer better results compared to Ray (Table VI).

TABLE VI:

		Ray		Spark	
Nodes	Dataset (GB)	R^2	RMSE	R^2	RMSE
5	2.2	0.5902	0.3200	0.6023	0.3152
3	2.2	0.5810	0.3236	0.6023	0.3152
1	2.2	0.5833	0.3227	0.6023	0.3152
5	4.4	0.5951	0.3181	0.6023	0.3152
3	4.4	0.5974	0.3172	0.6023	0.3152
1	4.4	0.5776	0.3249	0.6023	0.3152
5	8.8	0.6015	0.3156	0.6023	0.3153
3	8.8	0.5815	0.3234	0.6023	0.3153
1	8.8	0.5813	0.3235	0.6023	0.3153

VIII. Comparison of Ray and Apache Spark

In this section we will provide a comprehensive comparison, of the two frameworks.

A. *Ease of Use*

We found that Ray's setup is very simple compared to Sparks' since it only involved installing the necessary packages and running a few commands to connect our VMs to the cluster. The distributed execution of our scripts was easily configurable in both frameworks. Apache Spark and Ray are designed to address different needs, leading to distinct development experiences. Spark provides a higher-level abstraction, making it well-suited for large-scale data processing and batch workloads, but it follows a more structured execution model with less flexibility in fine-grained task control. In contrast, Ray offers more dynamic and low-level control over distributed workloads, enabling efficient execution of machine learning and parallel computing tasks with built-in automation for scaling and task scheduling.

Ray simplifies parallelization in its code. In many cases, parallel execution was achieved by simply adding the *@ray.remote* decorator to functions, allowing seamless distribution across multiple nodes. In contrast, Apache Spark offers a more structured and user-friendly coding experience, particularly with its high-level APIs and built-in optimizations. While Ray enables distributed execution with minimal refactoring, this flexibility sometimes comes at the cost of missing out on Spark's optimized operations and specialized libraries, which are designed to enhance performance in large-scale data processing tasks. For example PageRank which has a distributed algorithm that is difficult to manually implement efficiently , was handled trivially through Spark's many libraries. When it came to machine learning tasks however, both Ray and Apache Spark were friendly to code in because both offer well known APIs for these tasks (sklearn and Mlib).

Both platforms offer great Dashboard support, however Ray Dashboard was more polished than YARN's.

B. *Performance*

Memory efficiency is a crucial metric for evaluating a framework's performance, especially when processing large datasets. Effective memory management enables the execution of complex models and large-scale data processing without system failures. A framework that optimizes memory usage not only improves performance with existing resources but also reduces operational costs by minimizing the need for costly hardware upgrades.

Ray Dashboard provides a real-time interface to monitor memory usage across all machines in a Ray cluster while a script is executing. This allowed us to observe memory consumption in Ray script executions. We can also observe the memory consumption of Spark scripts through the YARN dashboard.

In our experiments we noticed that Ray is quite memory-hungry during the ETL operations, and we suggest that it is the main bottleneck in the performance of the framework in those workloads. On the other hand, Spark surpassed Ray in memory consumption in the Machine Learning workloads perhaps indicating which operation is more suited for which framework. Ray suffered from a lot of Out of Memory Errors and required a lot of disk spilling to run some of the operations, further hindering its performance in those tasks.

It is time to analyze the execution runtime for each workload which measures how efficiently a framework completes tasks. The comparison between Ray and Apache Spark revealed that their performance varies depending on the operation:

Ray outperforms Spark in simpler machine learning such as k-means clustering and in more complex ones like Linear Regression prediction. Overall, Ray showed better runtimes scores in these tasks. However, Spark showed decent results in our key metrics ($R^2 and\ RMSE$).

For the graph operations both frameworks have very similar performances. We evaluated the performance in two key operations, Triangle counting and PageRank but neither frameworks seemed to edge out over the other.

For the ETL Tasks Ray struggled to keep up with Spark. Spark was immensely more powerful in these workloads and outperformed Ray by a big margin (in some operations we had 10x difference in performance). Apache Spark excels at intensive data analytics tasks.

C. *Scalability*

To assess the scalability of Apache Spark and Ray, we conducted experiments using datasets of varying types and sizes, along with different numbers of nodes in the cluster. This allowed us to evaluate how well each framework handles increasing data volumes and adapts to changes in computational resources.

Regarding data scaling which refers to a framework's ability to efficiently manage and process large datasets as their size grows, our experiments offered mixed results. Execution runtime varied between the two frameworks based on the task. However Spark does seem to handle larger datasets better since the difference in performance between the smaller and the largest datasets was not as big.

When additional computational resources were added while keeping the dataset size constant, our results showed that Spark exhibits strong scaling behavior. Spark seemed to leverage the distributed execution better and had significant impovements when we added nodes. Meanwhile Ray did not seem to scale well when more nodes were added in the Machine Learning Tasks. However in the ETL tasks Ray utilized the additional workers, reducing runtime for all operations and all dataset sizes.

IX. Conclusion

Our research conducted an in-depth comparative analysis of Apache Spark and Ray, evaluating their strengths and weaknesses across various machine learning and graph-processing tasks. The goal of this study is to help practitioners and researchers make informed decisions when selecting the framework that best suits their needs in real-world scenarios.

Our findings reveal that both frameworks offer ease of use, automatic handling of distributed tasks, and scalability,. However, Spark's structured execution model, high-level APIs, and optimized libraries make it more efficient for large-scale data analytics tasks, while Ray demonstrated significant advantages in dynamic workload distribution, Spark remains a more mature ecosystem with a vast library of tools specifically optimized for big data processing and machine learning.

Rather than viewing Spark and Ray as direct competitors, our analysis suggests they can be highly complementary when used together. Spark is highly efficient for large-scale data analytics, quickly processing massive datasets to extract valuable insights. Ray, on the other hand, is well-suited for distributed machine learning.

Both Spark and Ray offer impressive capabilities in big data analytics, machine learning,. While Spark provides a robust and optimized ecosystem for structured data processing, Ray offers greater flexibility for distributed computing and real-time machine learning workloads. Understanding their respective strengths and how they can complement each other can significantly enhance a project's scalability and performance in the era of Big Data and AI.

REFERENCES

[1] N.Kassaris, K.vougias, "NTUA-BigData-Spark-Ray," github.com. https://github.com/NtinosVg/NTUA-BigData-Spark-Ray.

[2] Okeanos-Knossos, Greek Research and Technology Network (GR-NET), okeanos-knossos.grnet.gr. https://okeanos-knossos.grnet.gr/home/ (accessed: Feb. 4, 2025).

[3] Apache Software Foundation, "Apache Hadoop," apache.org. https:

//hadoop.apache.org/ (accessed: Feb. 4, 2025).

[4] D. Borthakur, "HDFS Architecture Guide," apache.org. https://hadoop. apache.org/docs/r1.2.1/hdfs design.html (accessed: Feb. 4, 2025).

[5] "Overview — Ray 2.34.0," docs.ray.io. https://docs.ray.io/en/latest/ray-overview/ (accessed: Feb. 4, 2025).

[6] Apache Spark, spark.apache.org. https://spark.apache.org/docs/latest/ (accessed: Feb. 4, 2025).

[7] "How Can I Access All My VMs Using One Public IP (NAT)?," okeanos-knossos.grnet.gr. https://okeanos-knossos.grnet.gr/support/user-guide/ cyclades-how-can-i-access-all-my-vms-using-one-public-ip-nat/ (accessed: Feb. 4, 2025).

[8] konect.cc http://konect.cc/ (accessed: Feb. 6, 2025).

[9] snap.stanford.edu. https://snap.stanford.edu/data/ (accessed: Feb. 5, 2025).

[10] Ray 2.35.0 docs.ray.io. https://docs.ray.io/en/latest/data/performance-tips.html (accessed: Feb. 8, 2025).