Python Scaling Framework Comparison in Big Data and Machine Learning Applications: Ray vs Spark

Pyliotis Athanasios
Electrical and Computer Engineering
NTUA
Athens, Greece
athan.pyl@gmail.com
el19050

Chourdaki Ioanna
Electrical and Computer Engineering
NTUA
Athens, Greece
ioannaxourdaki@gmail.com
el19208

Kefallinos Dionysios
Electrical and Computer Engineering
NTUA
Athens, Greece
dion397kef@gmail.com
el19030

***Abstract*—With the sheer volume of data available today for processing, the bottleneck for performance is often the management of the available hardware resources. Python frameworks such as *Ray* and *Apache Spark* aim to make this process as efficient as possible, and facilitate the scaling of data heavy workloads such as *Extract, Transform, Load (ETL)* operations or *Machine Learning* applications in distributed computing environments. The focus of this work is to compare the aforementioned frameworks in terms of performance, efficiency and scaling capabilities in the context of different data intensive workloads, running in a distributed computer network, and showcase their comparative strengths and weaknesses in these different scenarios.**

***Index Terms*—Big Data, Machine Learning, Python, Scaling Frameworks, Ray, Spark, Distributed Computing, Cluster**

I. Introduction

As the amount of data stored in the world rapidly increases, the need for faster and better processing becomes necessary and at the same time creates huge opportunities in the Big Data industry. Fields such as Machine Learning and Artificial Intelligence rely on the scaling of their models to keep increasing their capabilities and the resulting profits. Even simpler applications such as basic query operations on massive databases require immense computing power to complete in a reasonable timeframe. This is why, in the last decades, distributed computing has become the norm when it comes to these tasks. Logically, it follows, that frameworks facilitating the efficient distribution and management of the operations in these networks have become extremely important and their improvement of paramount significance.

There are numerous Python Libraries and Frameworks created for this purpose, some focusing on something specific while others covering a more diverse spectrum of needs. It is only natural that the question of which one to use in each specific usecase arises, and it is of huge importance. In this work we compare the frameworks *Apache Spark* [1] (referred to as *Spark* from now on) and *Ray* [2], both of which offer a variety of data management capabilities, while also having core differences which we will explore in the following section.

To compare the frameworks in a controlled and stable environment we used a cluster of 3 Virtual Machines hosted on Google Cloud Platform (GCP) [3]. The exact configuration setup will be explained in section III, and a detailed setup guide, as well as all the code developed for this project will be provided within the team's GitHub repository.

The applications tested will range in the following 3 categories, as suggested in the project description:

- Extract, Transform, Load Operations (ETL)
- Graph Operations (PageRank, Triangle Count)
- Machine Learning (ML)

Our observations following the testing phase and the results' evaluation are of huge significance for anyone trying to optimize their choice of framework in the specific scenarios but also extract a more general methodology for choosing how to manage a distributed network under certain conditions.

II. Software Layer

In this section we will take a closer look at the software components of this project, namely the constituents of the Spark environment; PySpark, HDFS and YARN, and also Ray and the APIs it comes with.

A. Spark Environment

Spark is a robust open-source framework designed for large-scale data processing and analytics. It provides a unified engine capable of handling diverse tasks, ranging from simple data transformations to complex Machine Learning operations. Spark's primary strength lies in its ability to distribute workloads across a cluster of machines, ensuring efficient and scalable processing. This capability is crucial for managing our workloads, which use a dataset larger than what a single machine could handle efficiently.

Under the hood, Spark uses *Resilient Distributed Datasets* (RDD), a programming abstraction that represents an immutable collection of objects that can be split across a computing cluster. Operations on the RDDs can also be split across the cluster and executed in a parallel batch process. Spark turns data processing commands into a *Directed Acyclic Graph* (DAG). When a Spark job is submitted, it is first broken down into stages based on the transformations applied to the data. Each stage consists of a series of tasks, which correspond to operations on partitions of the data. These

tasks are then distributed across the available nodes in the cluster. Spark's DAG scheduler optimizes the execution plan by determining the most efficient way to execute these stages, taking into account data locality and minimizing shuffling (i.e., the data transfer between nodes). By distributing tasks in this manner, Spark ensures that the workload is balanced across the cluster, maximizing resource utilization and achieving high performance.

Spark can be setup for a cluster in tandem with other software tools for efficient data storage and management of resources.

For this project, we configured Spark to run in cluster mode, leveraging YARN and HADOOP to store and manage resources efficiently. We installed Spark on each node of the cluster and configured it to interact with the Hadoop Distributed File System (HDFS) [4] for data storage and Yet Another Resource Negotiator (YARN) [5] for resource allocation:

- HDFS is simply a distributed file system that provides reliable and scalable storage for large datasets across multiple nodes in a cluster. It is designed to store very large files, spanning multiple terabytes, across a distributed architecture. HDFS ensures data redundancy and fault tolerance by replicating data blocks across different nodes, which allows for continuous operation even if some nodes fail. This makes HDFS a crucial component for handling the large-scale data processing required in this project. The replication factor (how many times the data are stored among the nodes) can be configured by the user, so in every application we chose to have it set to 2, in order to have sufficient redundancy without compromising too many system resources.
- YARN is a resource management layer within the Hadoop ecosystem that allocates system resources to various applications running in the cluster. It works by dividing the available hardware resources, such as CPU and memory, among the different tasks, ensuring that each application gets the necessary resources to operate efficiently. YARN handles job scheduling, monitors the execution of tasks, and manages the distribution of resources dynamically, which is essential for optimizing the performance of Spark jobs running in the cluster.

The configuration of the Spark work environment with HDFS and YARN allowed us to efficiently manage both data storage and computational resources, ensuring that the cluster operated at maximum efficiency. A comprehensive configuration guide for our work environment is included in the project GitHub repository.

Lastly, the PySpark API and many other Python libraries were employed to write the data processing tasks in Python, and then distribute and execute them across the cluster. Namely, `pyspark.sql` [6] for data manipulation, `GraphFrames` [7] for the Graph Operations part and `RandomForestClassifier` [8], `pyspark.ml.kmeans` [9], `tensorflow.keras` [10] and `ResNet50` [11] for Machine Learning were among the most used libraries. The use of these libraries in each task will be explored further in the following sections, while documentation for all of them is readily available in the project's References.

B. Ray

Ray is another open-source framework designed for distributed computing and scaling of Python applications. It provides a simple, flexible API that allows developers to parallelize and distribute workloads easily. Ray's architecture is centered around a distributed scheduler that allocates tasks across available resources, making it ideal for applications that require low-latency and high-throughput, such as machine learning training and hyperparameter tuning.

Ray's core abstraction is the *actor/task* model, where individual functions can be turned into distributed tasks. These *Tasks* can be deployed across a cluster, with Ray managing their lifecycles and communication, while the developer can specify their resource requirements. These specifications are then used by the cluster scheduler to distribute tasks across the cluster for parallelized execution.

Actors are an API extension of the basic Ray functionality, which allows the distribution of classes to the cluster instead of just individual functions. When an actor is instantiated he is essentially a worker and methods are scheduled on this specific worker and can use its resources (which can again be specified by the developer).

Ray supports creating ray-specific clusters by initiating them with a .yaml configuration file, or initiating a Ray session on an existing cluster just by denoting the head and worker nodes. For simplicity we used the second method, and setup our Ray cluster exactly like the Spark cluster (with two workers and one master node). Due to certain arising difficulties when integrating Ray with the HDFS service, we decided to store the dataset locally for this part of the testing. This did not impact the Speed of operations at all, since the time difference between reading from the HDFS or from local directories is negligible.

In this work, besides the basic `ray.data` [12] API, we used a plethora of different supported packages. Specifically, the use of `ray.train` [13] was necessary to facilitate the training of our models, and speed up the preprocessing part. We also used `xgboost` [14] for the Random Forest training, `sklearn.cluster.kmeans` [15] for the K-Means ML training, and `networkx` [16] for the Graph Operations. We also used the built-in `ray.remote` [17] decorator, extensively, for parallelization of non ray-native libraries. As with the respective Spark libraries, their use will be explained further at the following sections.

III. Hardware Setup

Due to the available packages that come with the Google Cloud Platform free trial plan, we had to setup an asymmetric cluster. We do not consider this a problem for our specific purposes, as long as the resources we have available can be utilized equally well by the two frameworks.

A. Specifications

Master:

- Ubuntu Server LTS 22.04 OS
- 2 Cores (4 VCPUs)
- 32GB RAM
- 50GB disk capacity

Worker-1 and Worker-2:

- Ubuntu Server LTS 22.04 OS
- 1 Core (2 VCPUs)
- 16GB RAM
- 50GB disk capacity

B. Connections

These machines are connected through the workers file, which is a file that has all the names of the workers of our cluster (master, worker-1 and worker-2 in our case) for Spark, and through the local IP Address of the master (10.0.168.2), which is considered the head node, for the Ray Cluster. All the installation commands, as mentioned in the installations part of our Github repository, need to be executed on all the nodes and then they should be connected and running smoothly. Because of the danger of cyber attacks we had to create a firewall rule, therefore all of our VMs are accessed through the public IP of the master node, only from machines connected to the NTUA network (147.102.0.0/16), through being physically there, or through the VPN service provided to all the members of the NTUA academic community.

C. Experiments

We ran experiments with 2 and 3 total VMs (referred to as "workers" in the results section) in each framework to compare their parallelization efficiency and scaling performance. In PySpark, we achieve this by configuring the cluster with the desired number of executor instances, while in Ray, we simply start the cluster with the number of nodes required for the experiment. For Spark the changes happen either by changing the number directly in the spark session or by providing it when executing the program, while for Ray we have created 2 initialisation bash scripts, one for 2 VMs in total and one for 3 VMs in total.

D. Google Cloud Platform

All of our VMs are hosted on the *Google Cloud Platform*. This decision was made due to the problem of cyber-attacks on the Okeanos-based VMs that were provided to us, which resulted to the loss of access to our machines. Google Cloud Platform is an interactive hosting platform that provides users with various different options according to their project's needs. For our cluster we chose the VMs mentioned above, all of which are E2-highmem type, a type of VM providing more RAM and disk memory, as we thought it would be better suited for Big Data applications. The master is E2-highmem-4 and the workers are E2-highmem-2, as Google Cloud Platform would not allow us to have more than 8 Virtual CPUs in our project. The differences can be found above in the *Specifications* subsection.

In order to connect to the cluster via SSH, we added our public keys in the metadata section for SSH keys and properly added the private keys of the machines in the `id_rsa` file of the `.ssh` directory of the user. As for the firewall, we used the built-in tool of Google Cloud Platform and simply added a rule to allow the ports that we use for HDFS (9870) and Yarn (8088) to be accessed only from Machines in the NTUA network.

IV. Dataset

In the selection of the dataset to be used for the ML tasks and the simpler ETL and graph operations, we considered mainly the factors of size and data types. We needed a significant amount of data, that could not fit in its entirety inside the cluster RAM, so the operations would have to be carried out using the resource managers of Spark and Ray respectively. The dataset also had to include numeric, string type and binary type data, as it was necessary for performing such a wide variety of tasks. We settled on using the NYC FHV (Uber/Lyft) Trip Data Expanded Dataset from Kaggle [18], which compiles a list of Taxi rides in New York City that took place between the years 2019 and 2022, and contains data about the duration, location, fare price, shared (pooled) ride requests, and other characteristics of these rides.

For different applications a different percentage of the data was used, as will be specified in every comparison that will be presented. The reason for this is mainly the timing constraints of training and using multiple models with such an expansive dataset, and also the storage constraints of our cluster setup. For future reference, the sizes of all used subsets of data can be found here:

Dataset sizes mentioned in this report:

- 4 months: 2.26 GB
- 6 months: 3.31 GB
- 8 months: 3.81 GB
- 1 year: 5.85 GB
- 2 years: 9.48 GB
- 3 years: 13.91 GB

In most applications the scaling of the data load was measured for 3 points; using 1, 2 and 3 years of data. This gave us the best possible intuition for the capabilities of the software as the load increases. However, due to the timing and resource limitations of our setup, we had to scale down our working set for some applications.

The NYC Trip Data dataset (hereby referred to as Taxi Data) consists of a main bulk of .parquet files, containing all the information of the rides, and a secondary index-volume named TaxiZones which matches the Location IDs of the pick-up and drop-off locations, used in the main volume, to their respective Location Name, Borough and Service Zone (larger location zone). This index was used mainly in the ML part of the comparison for feature extraction.

Additionally, a separate dataset was used for the more complex Machine Learning inference application that will be presented at the end, since it required images as input data. The dataset was found on Kaggle and is known as "CelebA" [19]. It

contains images of celebrities along with their characteristics. In our classification, we only used the image bytes and model ResNet50 from TensorFlow Keras, which was trained on the ImageNet dataset [20]. This decision was made due to the size of CelebA, which was a maximum of 1GB, and the classification happens to be to the closest item ResNet50 can identify the image as. In the end, we use 5-20 thousand images out of almost 200 thousand images for the classification, as our cluster could not handle more than that memory-wise.

To download the datasets to the cluster we took advantage of the functionality provided by the Kaggle API. After installing it with `pip install kaggle` and logging in our account, we used the `kaggle datasets download -d jeffsinsel/nyc-fhvhv-data` command to download the Taxi dataset locally, and afterwards we simply uploaded it to the HDFS. A similar procedure was used for the Images Dataset, except we did not have to upload this dataset to the HDFS as it would not make a difference on our comparison. For CelebA, we used the `kaggle datasets download -d jessicali9530/celeba-dataset` command and handled it locally.

V. Benchmarking

There were many different approaches to comparing the two frameworks, but after taking into consideration the assignment suggestions and the findings of our own brief research we settled on the following workflows for a thorough and varied comparison.

A. Applications

1) ETL: This is, of course, the most basic functionality every scaling framework designed for Big Data has to support. Extracting, Transforming and Loading data is required during the preprocessing stage of even the most complex tasks. A large model cannot be trained and provide useful service if the framework cannot go through the data efficiently, in order to convert it to a usable format.

When referring to ETL operations [21] we mainly refer to data manipulation that requires some form of loading part of the dataset (or the entire dataset), then executes some transformations or extracts information from the data, and finally returns the extracted information, or the dataset ready for use. A transformation can be simple, like a filtering of rows or some kind of sorting, but it can also be complex, like adding a new column with new information extracted from the same or a different table. These operations remind us of SQL database queries (as the name of the library `pyspark.sql` suggests), but their internal workings, especially when working on a cluster can be quite different, depending on how each framework represents data, how the driver manages resources and, of course, on how the methods are implemented in each library.

In this work we will start by examining 3 different ETL "queries". The first one will be a simple *sorting* method, and the second one will be a slightly more complex *filtering* of data followed by an *aggregation* function. Lastly, we will try a more complex *transformation* which extracts data from the rows and calculates a new datapoint to be stored in the table. It should be noted that this part of the comparison is not the only one where ETL plays a role. Some of the methods used during the preprocessing of ML features are more demanding and complex than the ones tested in the first part, but we think a straight-on, simpler comparison should precede them.

The only APIs used for this part were `ray.data` and `pyspark.sql` since they are the native options in each of the frameworks.

2) Graph Operations: Such operations are very commonly used in the fields of networks and systems [22], algorithms [23] and even mathematics and economics [24]. Since they are so commonly used, we decided to test the performance of our frameworks on them. This field includes famous algorithms like the shortest path algorithm, minimum spanning tree, graph connectivity, sub-graph counting/clustering and importance of nodes measurements. We ended up focusing on the last 2 areas and, more specifically, the algorithms of Triangle Count [25] and PageRank [26].

a) Triangle Count: is an operation that counts, for a given input graph, how many triangles (sets of 3 mutually connected nodes) are present in the graph. It is typically used for the analysis of the local clustering coefficient - a metric used to determine on what degree nodes in a graph tend to cluster together. It is used extensively in the network analysis field, especially in social networks of transitivity of connections, detecting communities and analyzing graph density.

b) PageRank: is an algorithm, developed by Google, used to rank web pages based on their importance in a web graph, which is why Google's search results are so good. It is also widely utilised in recommendation systems and social network analysis to rank entities based on their relative importance in a network.

Essentially, it assigns a ranking score to each node based on the number of incoming links (edges). It can be thought of as a measure of node centrality in a directed graph. The general formula for the rank of a node is:

$$PR(u) = \sum_{v \in B_u} \frac{PR(v)}{L(v)}$$

where u represents a page and v represents every page in the B_u set, which includes all the pages with links to u. $PR(v)$ is the PageRank value and $L(v)$ is the number of outbound links from page v. The initial PageRank (PR) for each website is calculated after we have all our equations from the above formula for each PageRank and then solve the equations system that we get.

For both of these implementations we found 2 possible libraries, `GraphFrames`, which is a Spark library and `Networkx` which is a library for computer network analysis in general. We decided to use the first one for the Spark implementation as it uses directly the dataframes of Spark to create the Graphs that we need and run the algorithms on them. Since Ray does not have a native library, we ended up

using the well-known library of `Networkx` to execute the algorithms and get our performance benchmarks.

3) Machine Learning (ML): ML tasks are essential benchmarks for evaluating the capabilities of distributed computing frameworks. These tasks typically demand considerable computational power, extensive data processing, and effective parallelization to train models efficiently on large datasets. Scalability is also a critical factor in ML workflows, as models must accommodate growing data volumes and increasingly complex algorithms. A truly scaling framework should be able to manage larger datasets and more resource-intensive computations without a noticeable decline in performance or a significant rise in resource usage. In this section, we evaluate the performance and scalability of Apache Spark and Ray by applying them to three different ML tasks: the *Random Forest* algorithm [27] for classification, the *K-Means* algorithm [28] for clustering, and a more complex process involving *Image classification* with a pre-trained model.

a) Random Forest: It is a supervised learning algorithm used for classification tasks. During training, it builds a collection of decision trees (”forest”), each one constructed using a different random subset of the data and a random selection of features at each split. This randomness helps prevent overfitting and enhances prediction accuracy. Each decision tree assigns a class to the data points it encounters. In the final step, the Random Forest combines the results from all the trees by using a majority vote to determine the class for each element. This approach ensures a more robust and accurate classification.

For the benchmarking, we implemented Random Forest classification in Spark using **MLlib**, specifically leveraging the `RandomForestClassifier` from `pyspark.ml.classification`. MLlib is a machine learning library within Spark, offering scalable algorithms and utilities for classification, regression, clustering, and recommendation tasks [29].

For Ray, we utilized **XGBoost**, a powerful machine learning library that is used for supervised learning tasks [14]. XGBoost is an implementation of gradient boosting, a technique that builds an ensemble of decision trees in a sequential manner to improve model performance. We accessed XGBoost via `XGBoostTrainer` from `ray.train.xgboost`.

b) K-Means: It is an unsupervised learning algorithm used for partitioning data into k distinct clusters. The algorithm first selects some random points as the starting centroids. Then, it assigns each data point to the nearest centroid, and the centroids are updated by calculating the average of all the points in a cluster. This process repeats until the centroids stabilize or a predefined number of iterations is reached. K-Means is useful for tasks like customer segmentation, and its Speed is particularly important when working with large datasets.

For Spark, we once again used **MLlib** by employing the `KMeans` from `pyspark.ml.clustering`.

In Ray, we utilized **Scikit-Learn** for K-Means clustering by importing `KMeans` from `sklearn.cluster` and parallelized the process using **Ray's remote functions**. In particular, we divided the dataset into chunks of size 1000 and created a Ray function named `kmeans_clustering` that utilized the Scikit-Learn K-Means algorithm and was implemented with `ray.remote` to enable parallel execution. Subsequently, we called this function for each chunk (using `kmeans_clustering.remote`) and gathered the final predictions (using `ray.get`). As a result, processing for each chunk occurs concurrently, effectively parallelizing the clustering task.

c) Complex ML: For the last ML workload, we decided to perform an *Image Classification* with a pre-trained model. For this puprose, we used the CelebA dataset [19] and the **ResNet50 model** of Tensorflow Keras [11] pre-trained on the ImageNet dataset [20]. In this workflow, first we load the data as Spark dataframes for Spark and Pandas dataframes for Ray. Then we perform the necessary preprossessing required by the model in order for the prediction to happen, using the included `preprocess_input` function from ResNet50. Finally, we classify the images based on their pixel HEX values. This Image classification task is more computationally intense, and was used for testing our frameworks in something more complex than simple model training.

B. Measurements

The comparisons presented in this work focus mainly on Speed differences between the two frameworks when it comes to the tasks of ETL, Graph Operations and ML. This is what will mainly be measured and analyzed. However, we also measured the memory footprint of most of the applications, using built in Python libraries or native Ray/Spark methods where it was deemed valuable for our analysis. During our measurements, we tried to be as precise as possible, so that we can extract the maximum intuition for the choice between the two frameworks.

When it comes to timing measurements, in each case we were interested in multiple parameters. First of all, the loading time for the datasets from storage was one thing that all workflows had in common, but it only took up a very small fraction of their runtime, so it is explored separately in the first section.

Secondly, on applications where data preprocessing was required, which is mainly on ML pipelines, we measured the time for it separately. This was to differentiate between the simple data manipulation capabilities and the training capabilities of the two frameworks separately, and produce more meaningful results.

Of course the metric we will be focusing on is the duration of the main task at hand, which might be the training of an ML model, the usage of an imported library component, or the manipulation of data using both frameworks. We will be analyzing this metric and calculating the *speedup* that would be gained from switching to the fastest framework in each task, while trying to interpret the result to the best of our ability, based on our knowledge of the inner workings of the frameworks.

Workers	Framework	Time (Seconds)		
		1 year	2 years	3 years
2	Spark	41.27	43.77	47.18
	Ray	20.33	21.51	24.97
3	Spark	41.42	45.72	47.76
	Ray	19.90	20.11	21.34

TABLE I
DATA LOADING TIMES

To define Speedup more clearly, since it will be used throughout the presentation of the results, we are using the following formula:

$$Speedup = \frac{Faster\ Framework\ Time}{Slower\ Framework\ Time}$$

For the timing measurements we utilized the Python `time` library, which has a simple enough API while being reliable and precise. For tracking the memory footprint of our applications we used the Ray Dashboard memory measurements and the `memory_usage` function from the `memory_profiler` library, which measures the application's memory usage throughout the program's execution. In ETL tasks, for simplicity, we experimented with `StageMetrics` from the `sparkmeasure` library that provides both memory and timing data.

All our measurements are included in the spreadsheet within the project GitHub repository, but we only present the most significant ones in this report.

VI. RESULTS

After designing the appropriate experiments, writing the scripts and running them on the cluster while measuring their performance, we now present the resulting measurements.

A. ETL

To set the expectations for this section, let's preface it by highlighting that Ray was mainly created to facilitate the deployment and scaling of large ML and AI dataloads, whereas Spark specializes in handling large-scale data analytics. Therefore it is obvious who has the advantage heading into pure ETL tasks.

1) Data Loading: In Table I we present the loading times for different dataset sizes with 2 and 3 workers, with both Spark and Ray. Ray seems to load data, and be ready to process them, faster in all configurations, probably because all the data is stored locally so no HFDS manager takes up any time overhead. This is not significant for our future measurements because data loading takes up a very small fraction of the total execution time in all configurations.

What might also be noticed is that increasing the number of workers seems to have a negative impact on the loading times of Spark. This might be due to the stored data blocks having to be distributed from the HDFS to more workers and the overhead getting bigger.

Something we noticed throughout this project, is that the loading times, especially with Spark, tend to show small fluctuations. Reading the same dataset multiple times we notice a timing difference of up to 30% between the runs. Our working theory is that this is related to where the data blocks end up being stored on the HDFS at the time of calling the script, since we know that the dataset is broken down into blocks by the cluster manager to be stored and mobilized more efficiently. Still, these loading time discrepancies are relatively small and do not account for any significant upsets in any of our measurements.

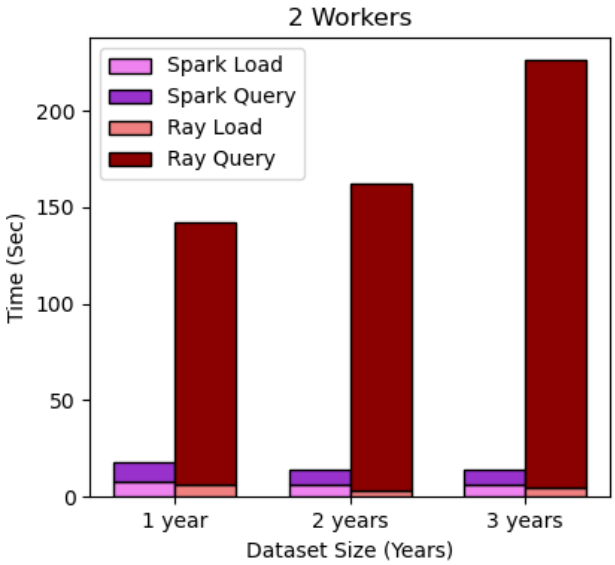

Fig. 1. ETL Sort - 2 Workers

2) Sorting: Our first operation on the Taxi Data dataset is simply sorting all the rides (each row has data for one ride) by ascending order depending on their duration. For this simple query, as mentioned before, we used `ray.data` and `pyspark.sql` from the two frameworks respectively. As can be observed in Fig. 1 the actual processing time difference for the same dataset size is severe, with Ray being on the back foot, especially as the load increases. However, due to the way we read data on the Spark script, the data loading time for Spark remains comparatively high. This is because on Spark we read all the columns of the dataset and select only the ones we are interested in. With Ray we can select the columns we want to read from the beginning, easing the strain both on the loading phase and the actual processing phase. Despite of this, the difference is still very much in favor of Spark.

In Fig. 2 we see the same comparison this time for 3 workers on the cluster, and besides the obvious decrease in time, since we have more processing power, we again notice a discrepancy in the loading times, especially in Spark, which has been seen and explained before.

It should also be noted that as the number of workers increases, the difference between the frameworks becomes slightly smaller. This might not be significant in such a small cluster, but it can already be seen that in larger configurations Ray starts to scale more efficiently.

On the other hand, an increase in data from 2 to 3 years of

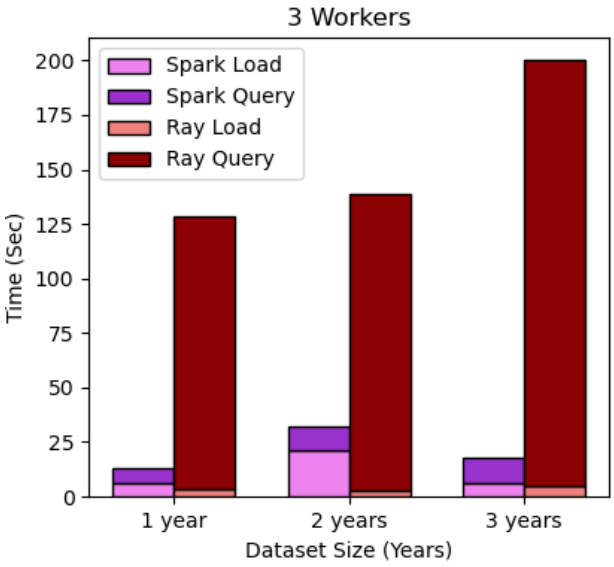

Fig. 2. ETL Sort - 3 Workers

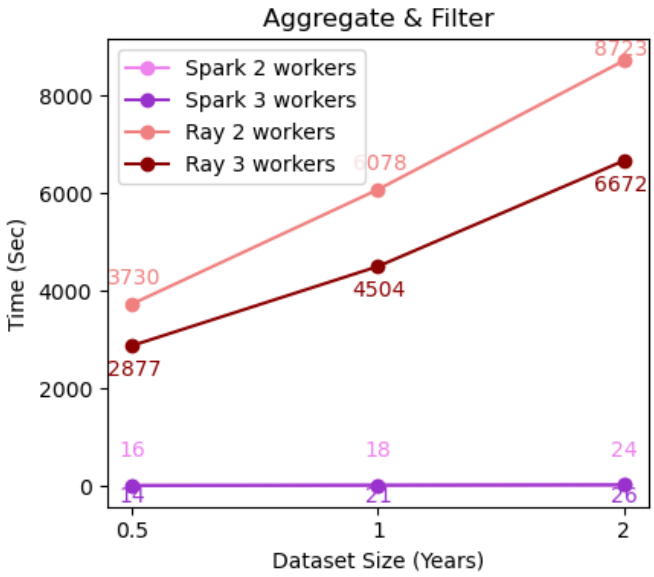

Fig. 3. Time Performance comparison of ETL Aggregate & Filter

Taxi Data (13.91GB at 3 years) seems to cause a lot more trouble for Ray, leading us to believe that Spark is more suited towards larger datasets, at least for this application and configuration. For now, we shall refrain from drawing general conclusions until we have a better picture of the results.

3) Aggregation & Filtering: This query involves keeping only the rides that dropped off passengers in Newark Airport, and then calculating the mean time from each of the 263 locations on our dataset to Newark. To achieve this we use the built in functions of `ray.data` and `pyspark.sql` to, initially, filter the rows that have Drop-off Location ID equal to 1 (which corresponds to Newark Airport) and then group the rows by their Pick-up Location ID, and aggregate the `trip_time` of the grouped data using the `avg` or `mean` functions to get the final result.

This operation really showcases Ray's inefficiency when it comes to such operations. The built-in grouping function is especially slow, according to our testing. As can be seen in Fig. 3 and both on 2 and 3 workers Spark took only a few seconds to finish, while Ray required half an hour at the very least for processing only the first year of data (5.85GB).

This might be because Ray's grouping function requires all the data rows, on which it operates, to be loaded in its object memory. Because so much data needs to be shuffled around, it makes sense that the process is slow, especially when considering the fact that sometimes data has to be spilled into disk memory if it cannot fit within the RAM Object Memory, something that slows down the process significantly. We will explain more about memory management and spilling shortly.

4) Transformation: In this query, we calculate the average Speed of each ride, using the `trip_miles` and `trip_time` columns, and dividing them while converting minutes to hours, to get the Speed in miles/hour. This query has to access several columns of the table while creating a new one and performing

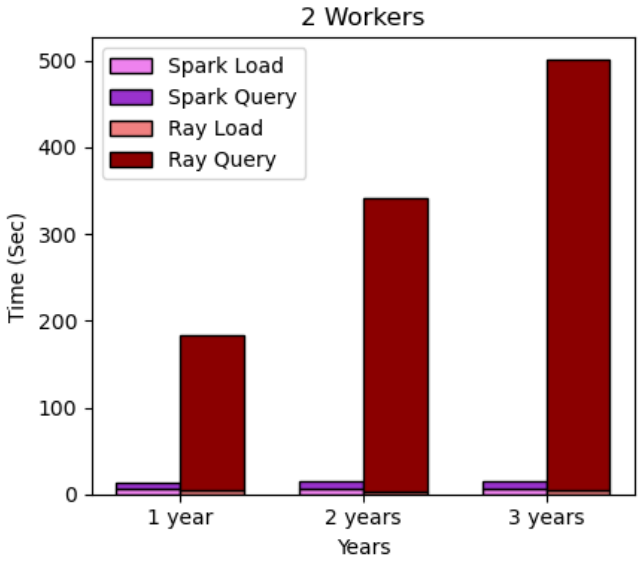

Fig. 4. ETL Transform - 2 Workers

calculations all at once.

The difference in performance, as can be seen in Fig. 4 and Fig. 5 is noticable here as well. As the size of the dataset increases, the time penalty of using the "wrong" framework (Ray) for this operation increases almost exponentially as can be seen in the Speedup plot of Fig. 6.

Increasing the number of workers, however, seems to have a much bigger impact on Ray than it does on the Spark application. This strengthens our assumption that Ray starts to scale more efficiently than Spark as the cluster increases in size, hinting to a better resource management technique.

Some notes on memory management: Comparing the

Fig. 5. ETL Transform - 3 Workers

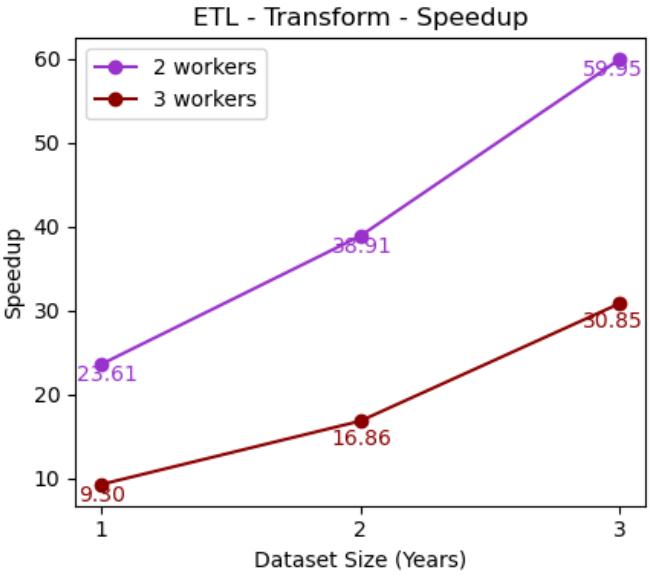

Fig. 6. Speedup of Transformation: Spark outperforms Ray

memory footprints of the ETL operations between the 2 frameworks is not only difficult but also pointless, because it is not a direct comparison. We can measure the total memory usage of the cluster for both operations (as we did) and we could see that Ray uses over 3 times the memory Spark uses (all results are contained in the spreadsheet). However, this result does not take into account the innate differences on how the two frameworks handle the reading of the data.

Spark uses cluster memory as efficiently as possible while Ray converts some of the available resources to object memory, no matter the task at hand. When the stored objects cannot fit inside the predefined object memory they spill to the disk, even though there might be RAM available on the cluster. This object spilling mechanism can impact performance since read-write time on disk is much slower than its RAM equivalent, and also can frequently cause Out-Of-Memory errors as we painstakingly discovered. Ray, as we will find out later, is more efficient when data streaming is happening, meaning objects go in and out of memory in a continuous pipeline and do not overload it.

B. Graph Operations

First, it needs to be mentioned that neither Ray nor Spark were created to perform graph algorithms, therefore there are considerable bottlenecks in their performance. For Spark, however, `GraphFrames` has been created to receive the data directly from the Spark dataframes, therefore it is expected to be a little faster. For Ray, we use the popular computer network library `Networkx`, which includes functions for graph operations, and is compatible with Ray. Ray and Spark were both created for parallelization of processes though, so we tried to leverage this functionality to improve our results.

1) Triangle Count: For this example we use the `PULocationID` and `DOLocationID` (pick up and drop off locations) columns and check how many triangles exist based on the paths that we have in our records. We execute our programs and get results for Time Performance and Speedup from each of the frameworks. As we can see from Fig.7, for this operation spark is exceptionally faster in comparison to Ray.

Based on Fig.8, we see that it is up to 117 times faster than Ray, and it is even faster for 3 workers than it is for 2 workers. Triangle Count is a relatively simple algorithm and, therefore, the reason why Ray may struggle with it may be because it's difficult for the framework to parallelize the execution of the `NetworkX` algorithm. Spark, on the other hand, seems to be able to do so for the `GraphFrames` algorithm. This shows the inability of Ray to handle such operations with currently existing built-in or imported functions.

2) PageRank: In the PageRank workflow, we use the algorithm again on the graph generated from the connections between the `PULocationID` and `DOLocationID` columns, in order to find out which destination is more central based on the records in our dataset. Fig.9 shows an advantage for Spark, as expected, but this time the difference is much smaller between the two frameworks.

Additionally, we can clearly see that the more we increase the size of the dataset, the slower both of them become, with Spark almost requiring the same time Ray needs and the Speedup being very close to 1 in Fig.10. It should also be noted that the increase in the number of workers, favors Ray once again.

Finally, checking the Peak Memory Usage in Fig.11 it is obvious that Ray requires more memory to perform the Pagerank Algorithm in comparison to Spark. We can also see that 3 workers use more memory than 2 in Ray, which makes sense since more memory is available on the cluster and therefore more memory is used as object memory.

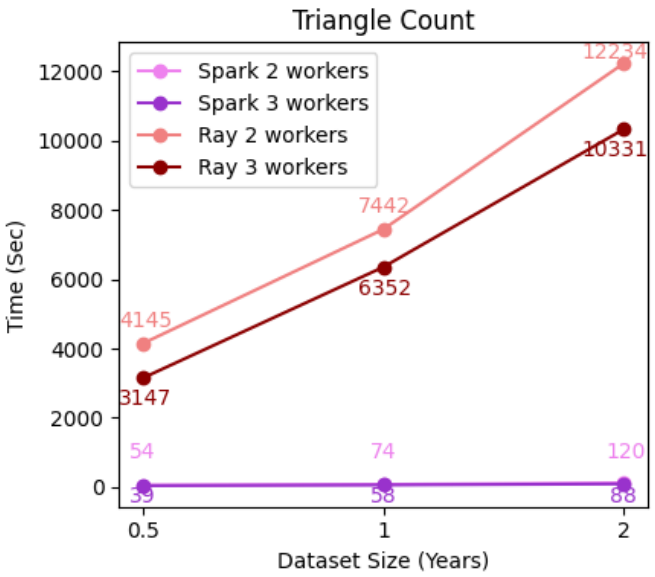

Fig. 7. Time Performance comparison of Triangle Count

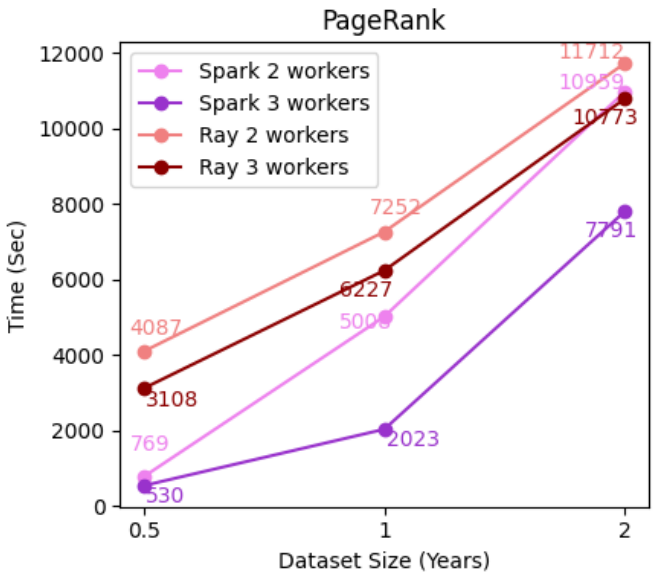

Fig. 9. Time Performance comparison of PageRank

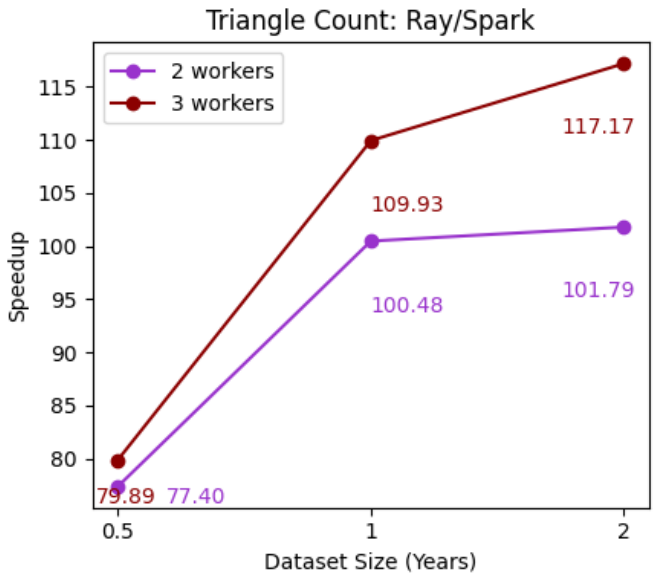

Fig. 8. Speedup of Triangle Count: Spark outperforms Ray

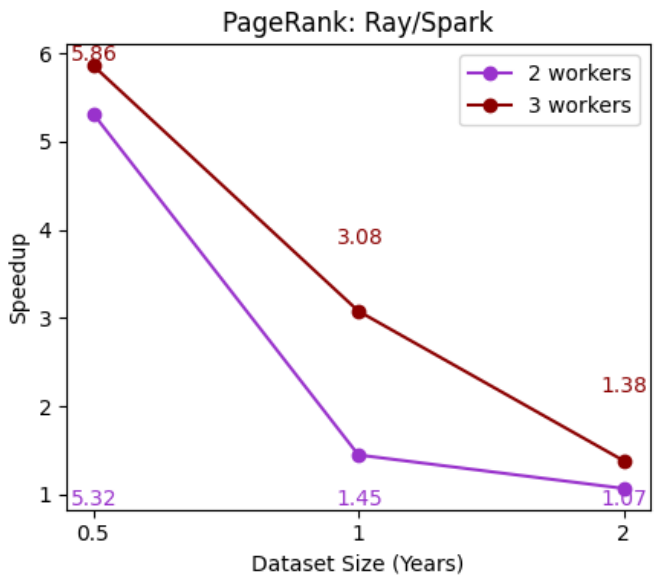

Fig. 10. Speedup of PageRank: Spark outperforms Ray

Conclusion on graph operations: Spark seems better suited for Graph operations in both Speed and Peak Memory Usage. For simpler algorithms Spark is up to 100 times faster, while for more complex ones like PageRank, the gap between their performance seems to be closing. Spark also handles memory better than Ray, at least for the Graph operations tested here.

C. Machine Learning

As mentioned above, Ray is a framework, mainly used for scaling AI and ML applications. It is designed to collaborate effectively with ML libraries such as Scikit-Learn and XGBoost, allowing users to scale their ML workloads easily. Given its focus on parallelization and optimized performance for ML-specific tasks, Ray is expected to outperform Apache Spark in the following tasks.

1) Random Forest: For this task, we tested different subsets of the Taxi Data dataset (4, 8, and 12 months) for 2 and 3 workers. During preprocessing, we map each `PULocation` and `DOLocation` to its respective Service Zone from the TaxiZones index table. We then select only the `PUZone`, the `DOZone` columns as features and the `Shared_Request_Flag` column as label, where `Shared_Request_Flag` indicates whether the passenger

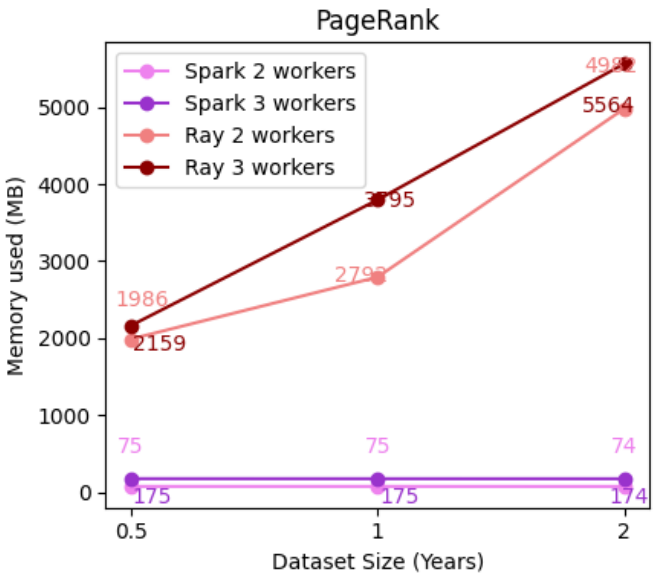

Fig. 11. Peak Memory Usage of PageRank

Fig. 12. Random Forest - 2 Workers

agreed to a shared ride. Finally, we remove any rows with `NaN` values. For the classification task, we utilize the Pick-up and Drop-off Location Zones to predict whether the `Shared_Request_Flag` should be classified as 0 (not a shared ride) or 1 (shared ride).

As demonstrated in Figures 12 and 13, Ray consistently outperforms Spark in overall execution time, demonstrating notably shorter training times. Based on Speedup observed in Fig. 14, Ray is up to 4 times faster than Spark. However, as the dataset size increases, the performance gap narrows, showcasing once more Spark's supremacy when it comes to larger datasets.

It is worth mentioning that Spark exhibits superior performance during the preprocessing phase. This performance discrepancy can be attributed to Spark's advanced optimization techniques for join operations. Spark uses various optimization techniques, such as broadcast joins and shuffle operations, which can significantly speed up joins, especially for large datasets. In addition, its Catalyst optimizer plays a crucial role in optimizing query plans, including join operations. In contrast, Ray does not have built-in support for join operations. The preprocessing in Ray involves a manual approach to mapping and merging data from two tables to join them, which is less efficient compared to Spark's automated and optimized techniques, resulting in slower performance.

When comparing performance with 2 versus 3 workers for the same dataset size, we observe that using 3 workers yields a 30%-50% improvement in performance for both frameworks, as expected.

The Peak Memory Usage shown in Fig. 15 is nearly constant for Ray, while in Spark, 3 workers perform significantly better than 2, indicating overall effective memory management.

2) K-Means: In this task, we tested different subsets of the Taxi Data dataset (1, 2, and 3 years) for 2 and 3 workers.

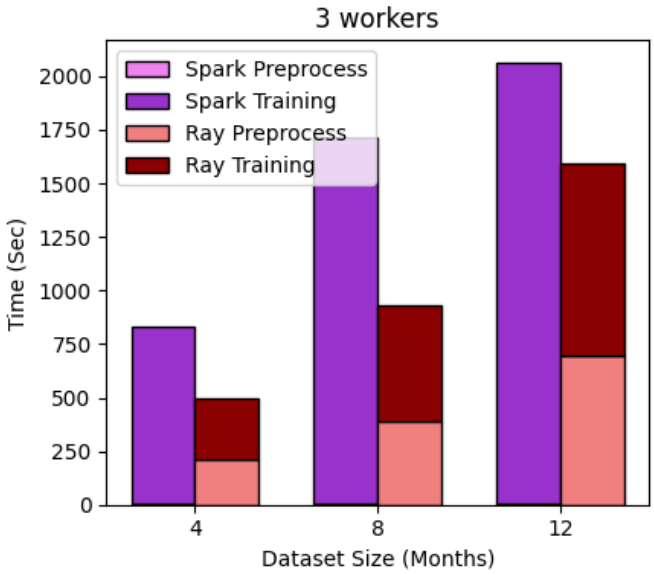

Fig. 13. Random Forest - 3 Workers

We selected Total Driver Pay and Total Trip Miles columns, removing any rows containing NaN values. Following that, we used those preprocessed columns as features for clustering, where we aimed to form $k = 3$ clusters: Cluster 0 (Low driver pay, Short trips), Cluster 1 (High driver pay, Long trips), and Cluster 2 (Medium driver pay, Medium trips).

As shown in Figures 16 and 17, the duration of training, as well as the comparison between different dataset sizes and configurations with 2 versus 3 workers, demonstrate results that closely align with those observed in the Random Forest (Fig. 12, 13). Accordingly, the observations provided before can be similarly applied in this context.

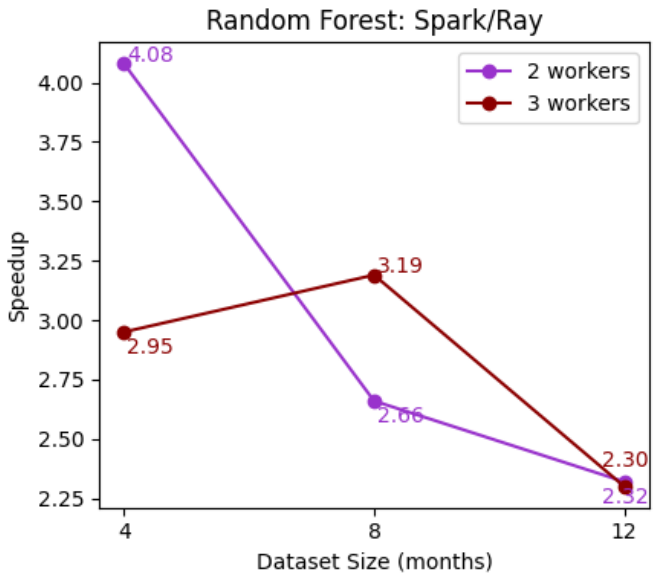

Fig. 14. Speedup of Random Forest: Ray outperforms Spark

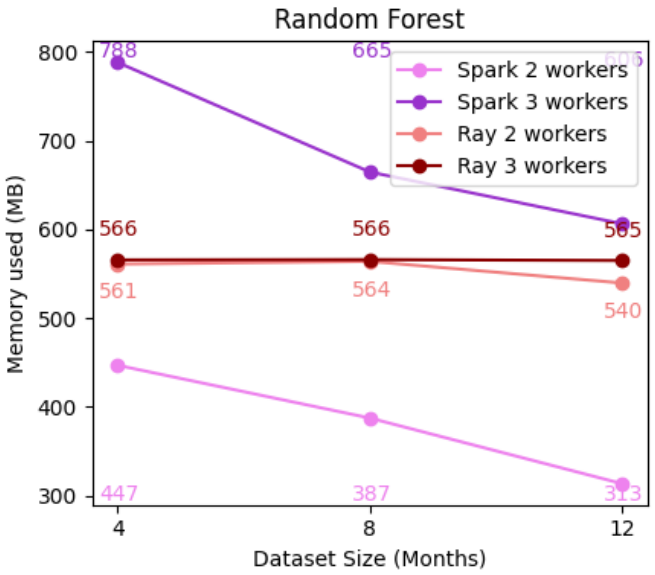

Fig. 15. Peak Memory Usage of Random Forest

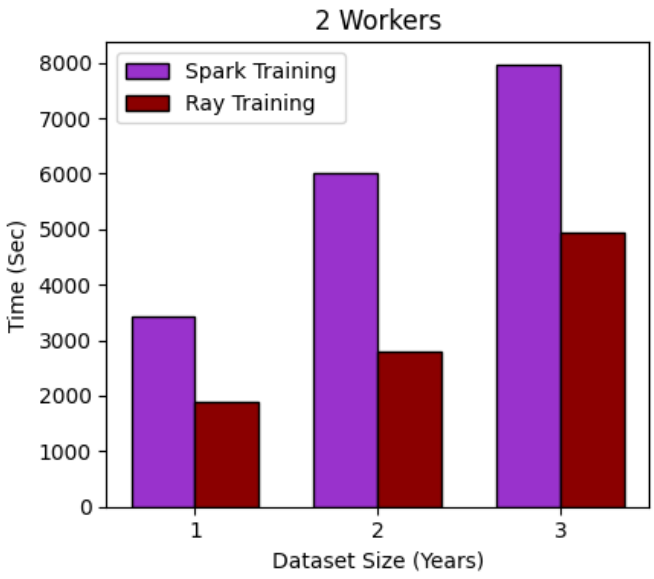

Fig. 16. KMeans - 2 Workers

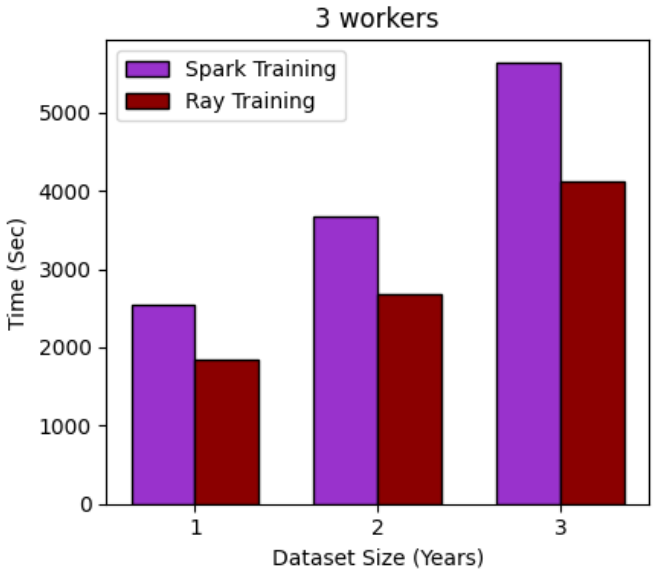

Fig. 17. KMeans - 3 Workers

Ray outperforms Spark once again, as expected (Fig. 18). However, with 3 workers, the performance gap between the two narrows to just 1.37 across all datasets, highlighting that in this particular case, Spark scales efficiently and nearly matches Ray, which is optimized for ML workloads. We also see Ray not struggling as much with increased loads, which indicates that the application might be sufficiently optimized.

In Fig.19, the Peak Memory Usage clearly shows that Ray consumes more memory than Spark when running the K-Means algorithm. When using Ray, the memory usage increases with 2 workers compared to 3, which is unexpected, as a larger amount of memory is accessible on the cluster with more workers. This might be because of slower processing times (when using 2 workers) keeping data in memory for longer periods of time, leading the peak memory usage to rise in this occasion.

3) Image Classification: For this workflow we are using the the *ResNet50 model* which is pre-trained on the *ImageNet* dataset [20] to detect the accessories worn by the celebrities in images from the *CelebA* dataset. After a little preprocessing we can feed the images to the model and get the timing results of Fig.20 and 21. Ray seems to be faster than Spark once again, but not by a mile. What should be noted here as well, is that as the number of worker increases, Ray seems to scale a lot

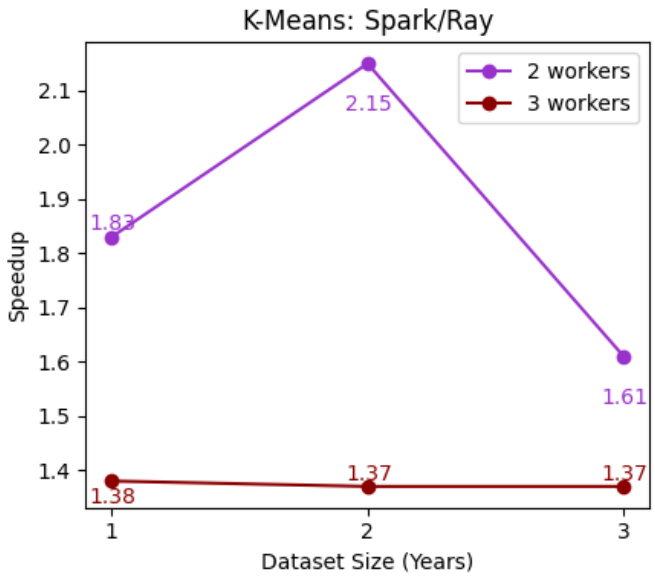

Fig. 18. Speedup of KMeans: Ray outperforms Spark

Fig. 20. Time Performance comparison of Image Classification

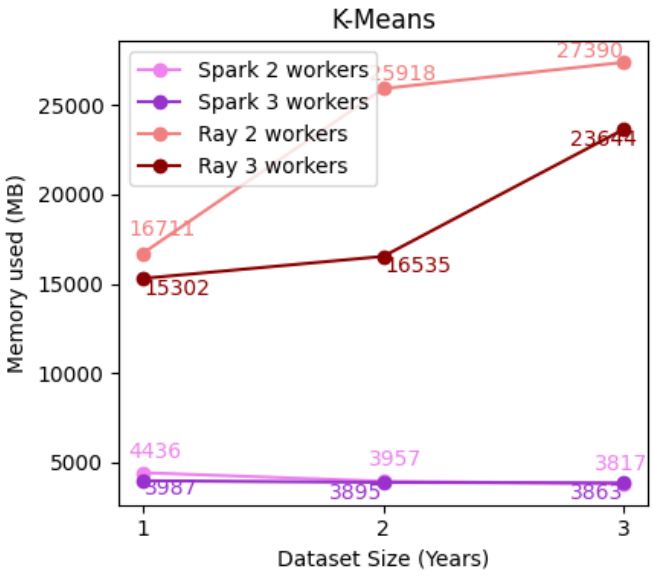

Fig. 19. Peak Memory Usage of KMeans

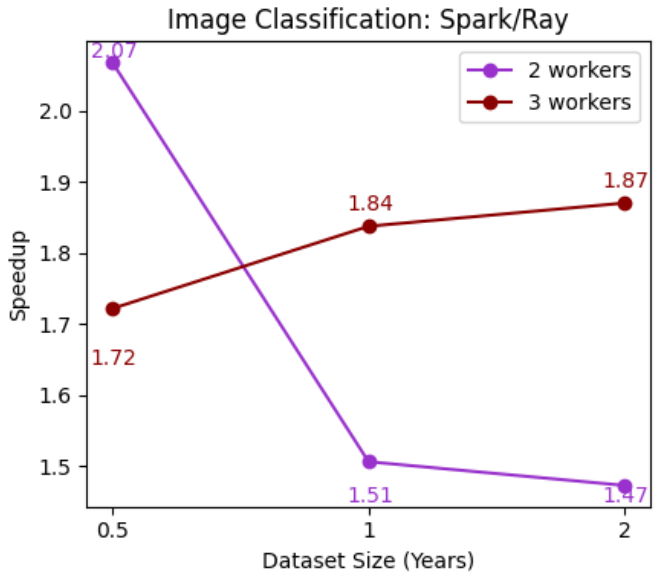

Fig. 21. Speedup of Image Classification: Ray outperforms Spark

better than Spark, which points once again to the fact that Ray is more specifically designed to scale pipelined workloads and behaves much better when it has more resources to manage.

Finally, when memory is considered in Fig.22 Ray still seems to require more the more workers it has, while Spark does not require nearly as much. This is the reason why Ray is not able to properly execute the code for more than 20k images, while Spark did not seem to have an issue with that. Once again Spark does a lot better when faced with the challenge of a growing dataset.

VII. Results Evaluation - Conclusion

After having completed several comparisons in different scenarios between Apache Spark and Ray framework, it is clear that no single framework holds a definitive advantage on the full spectrum of possible operations. What we extracted from this study is that when it comes to Big Data analytics it is hard to compete with the years of optimizations Spark has undergone, while being one of the most dominant frameworks in the field. The wide variety of ready-to-use functions, packages and libraries that integrate seamlessly with Spark is simply astonishing, and this means that in tasks such as data manipulation, graph operations or any task that requires lower

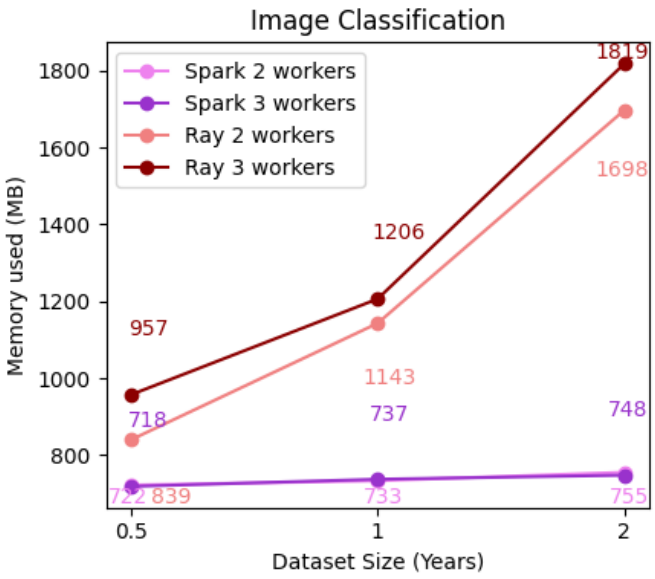

Fig. 22. Peak Memory Usage of Image Classification

level operations done on a massive set of data, Ray cannot be compared to Spark.

Spark also packs a lot of horsepower when it comes to model training and ML operations, with its prime advantage being the sheer volume of information it can efficiently process. Another secondary advantage it had over Ray, on which we did not comment previously, is how easy it was to setup and get working with different APIs, while having full support from years of documentation and troubleshooting available to help us.

On the other hand, Ray, as the newer competitor, focuses on one thing: scaling. We noticed a proportionally bigger improvement in Speed using Ray compared to Spark in almost every single task, when we increased the number of workers. This does not mean that Ray would certainly outperform Spark given enough workers, as that would depend on the nature of the task, but considering that it already outperforms Spark in ML model training and inference under certain conditions, we can be confident that Ray is the winner when it comes to scaling.

Of course, Ray has a lot of work to do, in order to get up to the standards of Spark, especially for amateur developers, as the available resources in terms of compatibility and documentation are very limited. All that said, we can be hopeful for Ray as it has already established itself as one of the emerging frameworks when it comes to scaling ML and AI workloads.

To sum things up one last time, if anyone was going to make a choice between using Ray or Spark for a specific *Machine Learning* workflow, we would recommend choosing Spark if the dataset sizes were large beyond a tipping point, and using Ray if they were mainly interested in scaling and deploying their application on a larger cluster, with continuous monitoring and resource sharing. However, when dealing with data manipulation on a large scale, Spark is king, also having the advantage of integrating with other Big Data frameworks easily.

We might also suggest that using both frameworks in tandem could be the optimal solution in some cases. Using Spark for preprocessing on the cluster, and delivering the features to Ray to be used for model training would absolutely yield the best results, better than using any framework exclusively.

VIII. Final Notes

Closing out this report, it should be noted that in some experiments we encountered inexplicable deviations from the expected results and, in some cases, results that could not be interpreted with our knowledge at the time. Many times we reran the tests, and sometimes the results changed (which was the most confusing), while other times they remained the same. In any case, the discrepancies we noticed were too small to shift our choice of framework in any specific scenario, even though sometimes we were unsure of how to explain them or include them in the presented plots. We tried our best to present data that made sense and explain the ones that did not without causing confusion or misinterpreting them. We hope this comparison helps at least some new users make a more informed decision in their choice of Python Framework.

IX. Acknowledgment

We would like to thank Professor Tsoumakos, for encouraging and suggesting projects that focus on researching real tools and getting hands-on experience in topics that span such a wide variety of useful and interesting fields of our science. We would never have the opportunity to dive so deeply into something that builds skills, some of which we do not even realize yet, if we stuck with the academic requirements of most courses.

References

[1] "Apache spark™ - unified engine for large-scale data analytics," https://spark.apache.org/.
[2] "Welcome ray! - ray 2.35.0," https://docs.ray.io/en/latest/index.html.
[3] "Google cloud: Cloud computing services," https://cloud.google.com/.
[4] "HDFS architecture guide," https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.
[5] "Apache hadoop YARN," https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html.
[6] "Spark SQL — PySpark 3.5.2 documentation," https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/index.html.
[7] "Overview - GraphFrames 0.8.0 documentation," https://graphframes.github.io/graphframes/docs/_site/index.html.
[8] "RandomForestClassifier — PySpark 3.5.2 documentation," https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.classification.RandomForestClassifier.html.
[9] "KMeans — PySpark 3.5.2 documentation," https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.clustering.KMeans.html.
[10] "Module: tf.keras," https://www.tensorflow.org/api_docs/python/tf/keras.
[11] "tf.keras.applications.resnet50," TensorFlow v2.16.1, available at: https://www.tensorflow.org/api_docs/python/tf/keras/applications/ResNet50.
[12] "Ray data: Scalable datasets for ML — ray 2.35.0," https://docs.ray.io/en/latest/data/data.html.
[13] "Ray train: Scalable model training — ray 2.35.0," https://docs.ray.io/en/latest/train/train.html.
[14] "Xgboost documentation," https://xgboost.readthedocs.io/en/stable/.

[15] "KMeans," https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html.
[16] "Algorithms — NetworkX 3.3 documentation," https://networkx.org/documentation/stable/reference/algorithms/index.html.
[17] "Ray.Remote — ray 2.35.0," https://docs.ray.io/en/latest/ray-core/api/doc/ray.remote.html.
[18] J. Sinsel, "NYC FHV (Uber/Lyft) trip data expanded (2019-2022)," https://www.kaggle.com/datasets/jeffsinsel/nyc-fhvhv-data, Oct 2023.
[19] J. Li, "Celebfaces attributes (celeba) dataset," https://www.kaggle.com/datasets/jessicali9530/celeba-dataset, 2018.
[20] "ImageNet," https://www.image-net.org/.
[21] S. H. A. El-Sappagh, A. M. A. Hendawi, and A. H. El Bastawissy, "A proposed model for data warehouse etl processes," *Journal of King Saud University - Computer and Information Sciences*, vol. 23, no. 2, p. 91–104, Jul. 2011. [Online]. Available: http://dx.doi.org/10.1016/j.jksuci.2011.05.005
[22] F. M. Atay and T. Bıyıkoğlu, "Graph operations and synchronization of complex networks," *Physical Review E*, vol. 72, no. 1, Jul. 2005. [Online]. Available: http://dx.doi.org/10.1103/PhysRevE.72.016217
[23] J. van LEEUWEN, *Graph Algorithms*. Elsevier, 1990, p. 525–631. [Online]. Available: http://dx.doi.org/10.1016/B978-0-444-88071-0.50015-1
[24] N. Yadav, I. Khan, and S. Grover, "Operational-economics based evaluation and selection of a power plant using graph theoretic approach," *Int J Energy Power Eng*, vol. 3, no. 4, pp. 249–259, 2010.
[25] T. Roughgarden, "Cs167: Reading in algorithms counting triangles," 2014.
[26] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999, previous number = SIDL-WP-1999-0120. [Online]. Available: http://ilpubs.stanford.edu:8090/422/
[27] L. Breiman, "Random forests," *Machine learning*, vol. 45, pp. 5–32, 2001.
[28] K. Krishna and M. N. Murty, "Genetic k-means algorithm," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 29, no. 3, pp. 433–439, 1999.
[29] Apache Spark, "Mllib," https://spark.apache.org/mllib/.