# A Comparative Study of Ray and Apache Spark in Big Data and Machine Learning

Ioannis Protogeros
*National Technical University of Athens*
03119008 || jprotog@gmail.com

Chrysa Pratikaki
*National Technical University of Athens*
03119131 || chrisaprat@gmail.com

Theodoros Papadopoulos
*National Technical University of Athens*
03119017 || paptheop01@gmail.com

***Abstract*—As the demand for processing vast amounts of data continues to surge, modern big data processing frameworks become increasingly important. This project focuses on comparing two specialized systems, *Ray* and *Apache Spark*, in the context of Extract, Transform, Load (ETL) tasks and both simple and more complex ML/AI workloads within the realm of big data projects. For our experiments, we generate and find suitable datasets of varying sizes that we store in a Hadoop Distributed File System (HDFS) across multiple machine nodes. Our aim in this work is to determine each of the two systems' relative strengths and weaknesses in these big data operations by monitoring their efficiency, scalability, and robustness across different workload sizes and available resources. Through our experimentation with the two systems, we conclude they demonstrate impressive capabilities in complementary Big Data Workload aspects, rendering them both powerhouses that facilitate these workloads; Apache Spark performs excellently in the handling of ETL operations on huge amounts of data, and Ray revolutionizes Distributed Machine Learning by integrating *Ray Datasets* with modern high-performing ML frameworks.**

***Index Terms*—Ray, Apache Spark, Distributed Computing, HDFS, ETL Pipelines,**

## AVAILABILITY

*Project Github Repository*
*Benchmark Results Spreadsheet*

## I. INTRODUCTION

### A. HDFS and Hadoop YARN

Hadoop is an open-source distributed processing framework that manages data processing and storage for big data applications. The Hadoop Distributed File System (HDFS) is the primary data storage system used by Hadoop applications. HDFS uses a primary/secondary architecture, employing a NameNode and DataNode architecture to implement a distributed file system. The NameNode keeps the directory tree of all files in the file system and the DataNodes are used for storage, where the data is broken down into separate blocks and distributed among the cluster. HDFS is designed to be highly fault tolerant, portable across a variety of hardware setups while allowing scalability across the cluster. Hadoop YARN is a resource management component of Hadoop which manages resource allocation and scheduling across the entire cluster efficiently. In our work, we will use the HDFS so that we will be able to store large datasets in a distributed manner, while Hadoop YARN will serve as the resource manager in our Apache Spark experiments.

### B. Ray

Ray is an open-source unified framework for scaling AI and Python applications like machine learning. Ray's unified compute framework consists of three layers, Ray AI Libraries a toolkit specifically for ML applications, Ray Core a general-purpose distributed computing library and Ray Clusters for cluster deployment.

Ray's execution architecture employs a dynamic task graph computation model, meaning that each application evolves during execution as a graph of interdependent tasks. Ray's programming model implements both a stateless process execution approach (task based) as well as a stateful one (actor based). When a task (i.e. remote function) is being executed it is automatically assigned to a set of Ray workers. Workers execute tasks serially, with no local state maintained across tasks. On the contrary, actors have to be explicitly instantiated and execute only when invoked [10].

TABLE I: Ray Task - Actor Comparison [10]

| **Tasks (stateless)** | **Actors (stateful)** |
|---|---|
| Fine-grained load balancing | Coarse-grained load balancing |
| Support for object locality | Poor locality support |
| High overhead for small updates | Low overhead for small updates |
| Efficient failure handling | Overhead from checkpointing |

Inside a Ray cluster, each node is equipped with a component called Raylet, which serves as the node's process manager. Raylets consist of two components, an object store and a task scheduler, and are shared between the jobs of each cluster node. The Raylet's object store takes care of memory management and ensures that processes have access to the data they need. It has shared memory across the node, so that each process has easy access to it. Every node is equipped with an object store, and all object stores collectively form the distributed object store of the cluster. The second component of the Raylet, the scheduler, mostly serves as the resource manager of each node. It acquires information about the number of CPUs and GPUs and the amount of memory available on its node, but it is always possible to register custom resources [12].

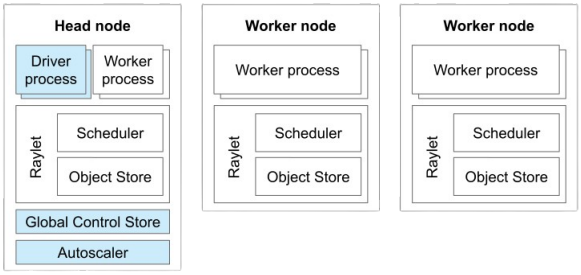


Fig. 1: Ray Cluster Overview

As mentioned, Ray provides a set of Core and AI Libraries specifically targeted for distributed data processing and ML applications. For this project, we will use Ray Data for scalable data processing. Ray Datasets are format agnostic, meaning that they can be used to train both PyTorch and Tensorflow models and offer a variety of compatible file formats (CSV, Parquet). We will also use Ray's PyTorch and XGBoost Trainers from Ray Train for end-to-end scalable distributed model training pipelines.

Lastly, we aim to use Ray Tune for tuning our ML pipelines. Ray Tune is a unified framework for model selection and training, which provides a narrow-waist interface between training scripts and search algorithms. Tune implements a user API for users seeking to train models, which will be used in this project for hyperparameter tuning, as well as a scheduling API, aimed at researchers to target a diverse range of workloads. Tune's API integrates many Hyperparameter search algorithms, such as two versions of the HyperBand algorithm, which will be used in this paper for training optimization [6], [18].

### C. Apache Spark

Apache Spark is an open-source unified analytics engine for large-scale data processing. Spark provides an interface for programming clusters with implicit data parallelism and fault tolerance. PySpark is the Python API for Apache Spark. With Pyspark we can extend Python with a distributed collection data structure (Resilient distributed datasets - RDDs).

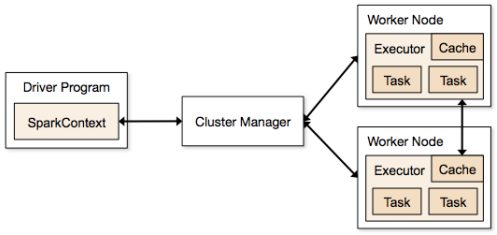


Fig. 2: Spark Cluster Overview

***Spark Core and Datasets:*** Spark Core runs over diverse cluster managers including Hadoop YARN, as previously mentioned. Spark Core is built upon the Resilient Distributed Datasets (**RDDs**) abstraction, which represents a partitioned, read-only collection of records. RDDs offer fault-tolerant, parallel data structures, allowing users to store data on disk or in memory, manage partitioning, and manipulate it using a diverse set of operators. This facilitates effective data sharing across computations, meeting the demands of various workloads. Aside for the RDD API, Apache Spark has introduced several improvements for its data abstraction, one of these being the **DataFrame** API, which is part of Spark SQL. Its goal is to combine the benefits of both RDDs and Spark SQL's optimized execution engine. DataFrames have become the main data representation in Spark's ML Pipelines API.

***Spark ML:*** MLlib is Spark's scalable library for providing machine learning primitives. In this project we will used Spark RDDs as well as MLlib's models for data intensive tasks and ML model training and tuning. Spark's MLlib is divided into two main packages `spark.mllib` and `spark.ml`, which are mostly built on top of RDDs and Spark DataFrames respectively. While both packages come with a variety of common machine learning tasks, in this project we will mostly use `spark.ml` and Spark DataFrames for model training, evaluation and optimization. [15]

## II. Setup of the Systems

### A. Infrastructure Used

For this project, we use a cluster that consists of 3 Virtual Machines provisioned by Okeanos-knossos, each with the following characteristics:

- OS: Ubuntu Server LTS 22.04
- 4 CPUs
- 8 GB Memory
- 30 GB Disk
- Python 3.11.5 (miniconda3)

All of our VMs were connected via a private network, while one also had a public IPv4 address, also provided by Okeanos. The public IPv4 address was needed to access the web applications associated with Hadoop, YARN, Spark, Ray and Grafana. However, in order to prevent attacks to the public address, we used a firewall, so we couldn't directly access this address. Instead, we used SSH connections and Port Forwarding (Tunelling) to access the web services in a more secure way.

### B. Hadoop and Apache Spark Installation

We followed the *Hadoop Adv DB Guide* for the Hadoop and Spark installation. This mostly consisted of installing and setting up the environment for Hadoop and Spark. After this, it was a matter of two commands on the "master" node ('start-dfs.sh' and 'start-yarn.sh') to invoke the scripts that would set up the HDFS and the YARN resource manager on the whole cluster. Therefore, the installation of pyspark was the only thing needed to run Python applications on the cluster over Spark. The Spark history server was also set up to access via a web app the history data of the completed tasks.

### C. Ray Setup

For our tasks we used Ray Core, Data, Train and Tune, so we needed to install the necessary Python packages to all machines with 'pip install ray[data,train,tune]'. After that, we

could configure our cluster consisting of three nodes, one head node and three worker nodes, The head node still containing a worker process in our tasks. The head node will run some additional control processes (driver process, global control store, and the autoscaler) [17]. We start the head node Ray process, and thus initiate the cluster by running the following command:

```
CLASSPATH=`$HADOOP_HOME/bin/hdfs \
classpath --glob` \
ray start --head \
--node-ip-address=192.168.0.4 \
--port=6379
```

Then we connect the workers to the cluster with the command

```
CLASSPATH=`$HADOOP_HOME/bin/hdfs \
classpath --glob` \
ray start \
--address='192.168.0.4:6379' \
```

In both commands we need to set the 'CLASSPATH' environment variable, and we also need to do it when we run a Python script on the Ray cluster. Both were needed in our case for the communication with the HDFS, from which all the nodes are able to read data.

Lastly, we follow the instrunctions [14] in the Ray documentation to set up Prometheus and a Grafana Server in the head node, so we can monitor our Ray applications' behaviour. It should be noted however that for larger tasks we preferred to not run these servers since the memory utilization on the head node was already high from the background running processes and as we will show during our study Ray is quite memory intensive.

*Additional Python Requirements*

1) **Pyspark**: `pyspark==3.3.2`
2) **Ray**: `ray==2.9.0`
3) **PyTorch**: `torch==2.1.2`
4) **XGBoost**: `xgboost==2.0.3`
5) **NetworkX**: `networkx==3.2.1`
6) **Librosa**: `librosa==0.10.1`
7) **Kaggle**: `kaggle==1.6.3`

## III. Data Discovery and Generation

For our purposes, we use a script that generates synthetic mixed data (both numeric and categorical features) and exports it in CSV format, using the `make_classification` function from the `sklearn.datasets` library. This is particularly useful because it allows the creation of CSV files with any desired size for benchmarking purposes. The script can be found at `data/data_generator_stdout.py`. After being generated, the data is finally put in the Hadoop Distributed File System for use across the cluster. Optionally, we stream the contents of the CSV file to the standard output and redirect (with the pipe '|' operator upon calling the Python script) to the 'hdfs put' command, so there is no need to write a large amount of data to disk before it is put in the hdfs.

For our ETL and ML scripts we use CSV files with sizes of around 2GB, 4GB and over 8GB, so that the entirety of the data won't be able to fit into main memory. These are datasets that contain 30, 60, and 120 million rows of tabular data respectively, both numeric (double floats) and categoric (integers, string objects). Table III gives an overview of the datasets used.

We also use graph data in the form of a list of edges to test several graph operations that will be described below. We both generate artificial data and discover real graphs. For the generated graphs we create a script that utilizes the NetworkX library to create Small-World graphs, which resemble many real-world social networks, using the Wattz-Strogatz model. This is again useful when we want to control the size of our graphs. We also use real graphs from the KONECT project [1], which provides many real-world graphs of various sizes and topologies, as well as useful metadata (degree centrality distributions, triangle amounts, clustering coefficient, etc.). Both artificial and real data are in the same format of TSV files. The real-world and generated graphs used were:

- 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 10k, 20k, 30k, 40k, and 50k and with an initial Regular Graph degree of 100, so each graph has 100 times more edges than vertices (i.e. the 30k node graph will have 3 million edges).
- Google hyperlinks (A network of web pages connected by hyperlinks. The data was released in 2002 by Google as a part of the Google Programming Contest. 875,713 nodes and 5,105,039 edges)
- 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 (es): 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. 3,033,373 nodes and 46,656,045 edges
- Wikipedia links (uk): This network consists of the wikilinks of the Wikipedia in the Ukrainian language (uk). Nodes are Wikipedia articles, and directed edges are wikilinks, i.e., hyperlinks within one wiki. 1,201,408 nodes and 59,136,839 edges

Lastly, we use a Kaggle dataset that is used for speech emotion detection. More specifically, we use the RAVDESS Emotional speech audio [7], or more accurately a subsection (1.2GB out of 24.8GB) of it that is available at Kaggle. To download it, after installing the Kaggle API ('pip install kaggle') we create an API token from a Kaggle account

TABLE II: Datasets used for Benchmarking

| Rows | #of CSV files | Total Size |
|---|---|---|
| 20 million | 2 | 2.011 GB |
| 60 million | 3 | 4.167 GB |
| 120 million | 6 | 8.333 GB |

TABLE III: Graphs used for Benchmarking

| Graph | nodes (n) | edges (m) |
|---|---|---|
| SW-$x$K | $x \cdot 1000$ | $100 \cdot n$ |
| google-web | 875,713 | 5,105,039 |
| lakes | 2,758,119 | 6,794,808 |
| higgs | 456,626 | 14,855,842 |
| wiki-es | 3,033,373 | 46,656,045 |
| wiki-uk | 1,201,408 | 59,136,839 |

and put it in '~/.kaggle/kaggle.json'. Then we download the dataset with 'kaggle datasets download uwrfkaggler/ravdess-emotional-speech-audio –path /data/ravdess –unzip' and put it in the HDFS with 'hdfs dfs -put'.

## IV. Tasks for Evaluating the Systems

### A. Graph Operations

***PageRank:*** PageRank is an algorithm used by Google Search to rank web pages in its search engine results. The fundamental idea behind PageRank is to assign a numerical value (rank) to each web page in a way that reflects its importance on the web. The underlying assumption is that more important pages are likely to receive more links from other websites. The algorithm works by counting the number and quality of links to a page to determine a rough estimate of the website's importance. In our implementation we used an example script provided by Spark (that we modified), and we followed that implementation to recreate the same algorithm using Ray to evaluate how each system performs in that task that uses the Map-Reduce programming paradigm. That way we also gain insight for the process of writing code to be executed distributedly in both systems. The Map-Reduce logic we implemented relies on the fact that each node $y$ emits their "contribution" $PR(y)/out(y)$ to all the nodes x they point to $(y \rightarrow x)$. This is more eloquently explained in [3], [20]. It becomes apparent that programming in this style is much more suited for Spark than it is for Ray, and that is also reflected in the experiment results as we will later see. It must be noted that both are naive implementations of PageRank, in contrast to the optimized algorithm provided by Spark's GraphX library. We also test the performance of this implementation against the same graphs. Specifically we use the 'graphframes' library which provides an API for GraphX algorithms in PySpark.

***Triangle Counting:*** Counting the number of triangles in a graph is a common problem in graph theory and network analysis. A triangle in a graph is a set of three vertices that are pairwise connected, forming a complete subgraph of size 3. The count of triangles in a graph can provide insights into the connectivity and structure of the network. In our context, we pitted the two systems against this task in the following way. For Spark, we used the GraphX library which provides an implementation for a triangle counting algorithm. For Ray on the other hand, we use a programming style that is more in accord with the system's ability to parallelize Python code: We use the NetworkX library to create a Graph Object, that is then stored with 'ray.put()' into Ray's object store. That way, all the workers will have access to the graph object. Then, we use Ray Tasks to assign to each worker machine the counting of the triangles for an equal part of the graph's vertices. We use the NetworkX library again, which provides an implementation of the Triangle Counting algorithm and takes the list of nodes for which the passing triangles are computed as an argument. All that means that for multiple machines, the algorithm is executed concurrently for an independent portion of the same graph.

TABLE IV: Dataset Schema

| f_1 | f_2 | f_3 | cat_1 | cat_2 | word | label |
|---|---|---|---|---|---|---|
| float64 | float64 | float64 | int | int | string | (0,1) |

### B. Extract, Transform, Load Operations

We evaluate each system's performance across several ETL operations, which are the building blocks of defining tasks that rely on transformations/manipulations of Spark Dataframes or RDDs as well as Ray Datasets. We monitor the runtime, CPU execution time, and Peak Memory Usage for the following ETL tasks:

- Loading large amounts of data from a common source accross the machines (in our case, 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.
- Performing "Group By" and Aggregation operations. We refer to the dataset's schema in table IV to describe the specific task: In both Spark and Ray we will group the dataset by the 'cat_1' values and then find the sum of the 'f_3' feature across those groups.
- Sorting according to a certain column. Specifically, we sort the same dataset with both systems according to the values of column 'f_1'.
- Applying transformations on the dataset. The transformations we apply are the following: First, we add a column called 'new_feature' which contains for each row the values of $f_1^2 + f_2^2$. Then, we filter out the rows for which the length of 'word' is larger than the value of 'new_feature'.

We use the artificial dataset that was described in the previous section for these tasks, which is stored in the form of a folder with multiple CSVs in the HDFS. We convert the data from CSV to a Spark Dataframe and a Ray dataset for the evaluation of each system on these tasks. We measure these metrics on different dataset sizes, up to big enough that the dataset may not fit in the main memory of a single machine, as well as on a different amount of workers to examine the performance benefits from having more available cores and memory.

### C. ML Algorithms and Pipelines

We are particularly interested in evaluating the two systems' performance in common Machine Learning and Artificial Intelligence Tasks. 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.

*K-Means Clustering:* First, we will pit both systems against a simple ML task, which consists of classifying the same dataset from before into two clusters. This is an iterative algorithm that starts by defining two centroids and classifying each point to the closest centroid based on a metric (e.g. the Euclidean distance, which we used). After many iterations, this algorithm usually converges. This was done for 10 iterations in both systems with operations on Spark Dataframes/Ray Datasets, while the performance of the Spark MLLib usage was examined. For Ray, we followed the Spark example script for K-Means by doing the same transformations, and we confirmed it works as expected by using it on a subset from the generated subset, the result being shown in figure 3.

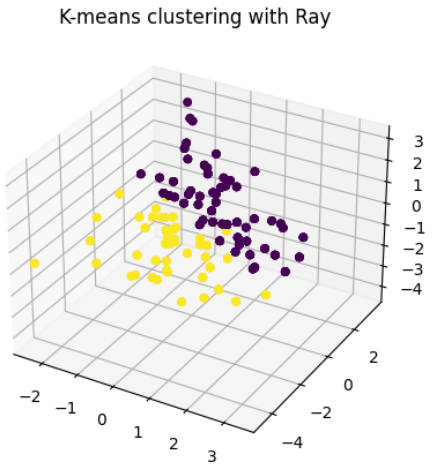


Fig. 3: Result of K-means clustering with Ray

***More Complex ML Workloads*** *- Data preprocessing and model training:* For a more end-to-end ML Pipeline, we define a task that consists of preprocessing our dataset and then executing some common supervised training algorithms to train models able to classify each row into one of the two classes. The preprocessing consists of the same tasks defined in the "transformations" part of the ETL tasks in subsection IV-B (note: we verified that the filter operation takes out a minimal number of rows out), as well as applying a MinMaxScaler and Vector Assembler/Concatenator transformation on the 'f_1,f_2,f_3' and 'new_feature' columns. Then, we trained a Multi-Layer Perceptron model as well as a Random Forest Classifier using both Ray Datasets' compatibility with PyTorch and XGBoost frameworks and Spark's MLLib implementation of an MLP and usage with XGBoost. For both systems and algorithms, we measure the runtime for preprocessing the dataset and executing the training algorithm.

*Hyperparameter Optimization:* Tuning the hyperparameters of a model is a famously time-consuming task since it consists of training a model multiple times with different hyperparameters and tracking its performance. We will attempt to take advantage of Ray and Spark's abilities to parallelize tasks like this and monitor their performance across performing a grid search on 4 different hyperparameter combinations for both MLP and Random Forests. For the Apache Spark tuning script, we used the `ParamGridBuilder` and `TrainValidationSplit` functions from the pyspark.ml.tuning library, while for the Ray script we used Ray Tune's `Tuner` module.

*Extracting Meaningful Features for Speech Emotion Detection:* **Librosa** [8], [9] is a powerful Python library that can help extract meaningful features for Speech Emotion Recognition, such as Mel-Frequency Cepstral Coefficients, chroma features, and spectral coefficients. These features can be used to train ML models to tackle the challenging task of recognising emotion in speech or music. We found this [4] Kaggle notebook that performs various tasks on the RAVDESS dataset, which include

- Exploratory Data Analysis (EDA)
- Data Augmentation
- Feature Extraction
- Testing several ML models on recognising speech emotion

As part of our study, we focused on the Data Augmentation and Feature Extraction parts of this notebook, to test how leveraging the abilities of Ray and Spark could enhance the performance of this time-consuming task. Hence, this task consisted of performing data augmentation by adding samples with noise and samples with stretch and pitch shift, essentially tripling the entire dataset, and then extracting the MFCCs of each audio file in the dataset. We aim to test each system's ability to accelerate this process. We used Ray's Tasks to call the function that performs data augmentation and extracts the features remotely (aka asynchronously) for each file, and we used Spark's User-Defined Functions (UDFs) for the same reason.

## V. Benchmarking - Evaluation

In this section, we will demonstrate the results of the benchmarks we ran on the systems against the tasks described in the previous section. All the tasks were executed across a different number of workers in the cluster (1,2,3) and across different sizes of the dataset (2,4,8 GB, as shown in table III). The metrics we examined in these tasks were **Runtime, CPU Time, Peak Heap Memory**. To extract these metrics from the tasks running across the cluster, we used SparkMeasure [5] for spark and the 'stats()' function for Ray Datasets [13], which provides useful metrics for the execution of different tasks upon a dataset. When needed, we also used the web application UIs that provided useful statistics (Spark history server, and Ray Dashboads's integration with Prometheus and

TABLE V: GraphX Triangle Counting Runtimes on Larger Graphs

| higgs | wiki-es | wiki-uk |
|---|---|---|
| 30.48s | 92.31s | 57.66s |

TABLE VI: PageRank Runtimes in Larger Graphs

| | Spark (naive) | Spark (GraphX) |
|---|---|---|
| **web-google** | 339.65 | 171.31 |
| **lakes** | 1485.44 | 1461.02 |

Grafana). Lastly, it is important to note that the metrics that are presented are the **mean results of the experiments being run multiple (at least three) times**, to ensure that the results are accurate and more robust - i.e. no outliers are being taken into account, and the impact of any fluctuation between runs is minimized.

### A. Graph Operations

*Triangle Counting:* In figure 4 we present the runtime statistics from running the triangle counting algorithm on our three-worker cluster in two ways:

- By parallelizing NetworkX's implementation using Ray, and
- By using Spark's GraphX library, which provides an implementation of the triangle counting algorithm.

It is evident that while the algorithm does benefit from the parallelization Ray provides (by delegating parts of the task to available workers), the implementation in Spark is more optimized and scales better with the size of the graph. We observed that the task of putting the Graph object into the object store introduces an overhead, although not significant. It should be noted that NetworkX uses Python Dictionaries to represent graphs (more specifically: dictionaries of dictionaries), which are costly as a data structure in terms of memory [19] and may become an issue with larger graphs. For reference, some results of GraphX's triangle counting on larger graphs are shown in table V.

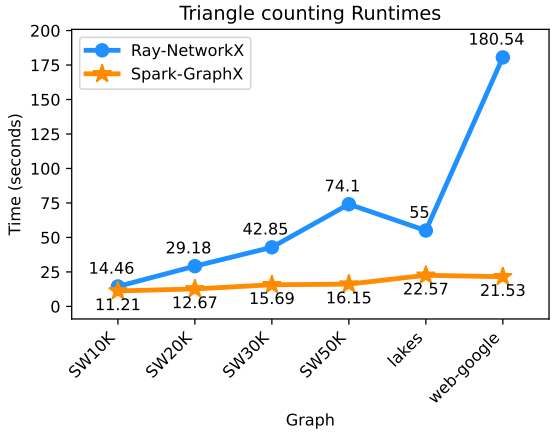


Fig. 4: Triangle Counting Runtimes

*PageRank:* We present in figure 5 the results of the Ray and Spark systems running the PageRank algorithm on the cluster, both with a "naive" side-to-side implementation that utilizes Spark Dataframes and Ray Datasets, and with GraphX's implementation. It is abundantly clear that Ray is far less suited for this task, taking more than 10x time to execute PageRank on all the examined graphs. For reference, we present some execution times for Spark in table VI. It should also be noted that programming the same task in Ray felt a lot more unnatural - There was not even a 'Join' operation implementation like in Spark, so a naive substitution by sorting and zipping the Ray Dataset had to be used.

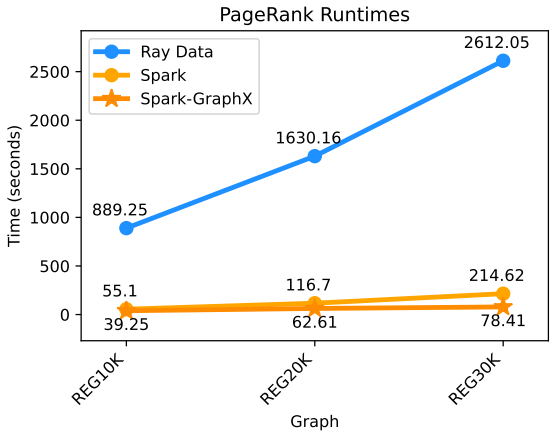


Fig. 5: PageRank Runtimes

### B. Extract, Transform, Load Operations

***Loading:*** In order to perform the load task we use the `spark.read.csv` method for Spark and the `ray.data.read_csv` method for the Ray script. Both frameworks perform adequately in this task on each dataset size, with Ray executing slightly faster as shown in table VII.

***Sorting:*** on the first feature of the CSV file: Both frameworks perform quite similarly on this task 6. However, Ray consistently fails to sort our largest CSV file (8GB) due to extreme memory (or disk, when spilling was required) usage. This is happening because sorting is a total exchange (all-to-all) operation and requires more memory (and disk spillage) to be successfully executed. It was observed during execution on the Ray cluster, that only one node's memory usage was increasing, even while executing sort on the entirety of the cluster, as shown in this screenshot 7 taken from Ray Dashboard's integration with Prometheus' monitoring stats and Grafana's visualization. We figure that this uneven memory load that was placed on one worker led to the out-of-memory errors. **Push-based shuffle:** Moreover, for the Ray Sort script to be successfully executed, the environment variable `RAY_DATA_PUSH_BASED_SHUFFLE` needed to be set. 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 operations like Dataset.random_shuffle, Dataset.sort and Dataset.groupby. Ray's documentation (see [16]) also states that shuffling can be challenging to scale to large data sizes, especially when the

TABLE VII: Collective Loading Benchmark Results

| Workers | Dataset (GB) | Spark Wall (s) | Spark CPU (s) | Spark Tasks | Spark Peak Heap Memory | Ray Wall (s) | Ray CPU (s) | Ray Tasks | Ray Peak Heap Memory |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 2 | 31.13 | 140.41 | 37 | 245 MB | 20.77 | 35.31 | 171 | 1759 MB |
| 2 | 2 | 40.30 | 145.36 | 37 | 211 MB | 22.75 | 37.77 | 171 | 1774 MB |
| 1 | 2 | 49.83 | 143.75 | 37 | 253 MB | 24.38 | 36.04 | 171 | 1729 MB |
| 3 | 4.167 | 49.53 | 340.14 | 93 | 240 MB | 24.28 | 59.76 | 63 | 1759 MB |
| 2 | 4.167 | 75.15 | 356.43 | 93 | 216 MB | 27.44 | 65.39 | 63 | 1745 MB |
| 1 | 4.167 | 129.82 | 348.18 | 93 | 230 MB | 31.42 | 60.28 | 63 | 1736 MB |
| 3 | 8.333 | 71.77 | 520.33 | 137 | 225 MB | 46.28 | 133.97 | 72 | 1799 MB |
| 2 | 8.333 | 80.44 | 485.7 | 137 | 227 MB | 48.07 | 147.24 | 72 | 1829 MB |
| 1 | 8.333 | 158.83 | 509.67 | 137 | 235 MB | 59.63 | 132.95 | 72 | 1815 MB |

total dataset size can't fit into memory, which justifies the fact that we couldn't apply the sort operator on the 8GB CSV file.

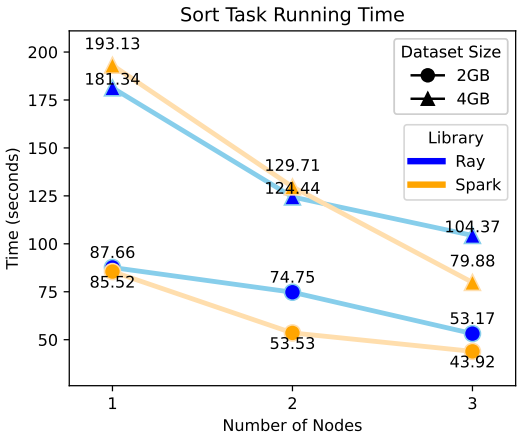


Fig. 6: Sort Running Times

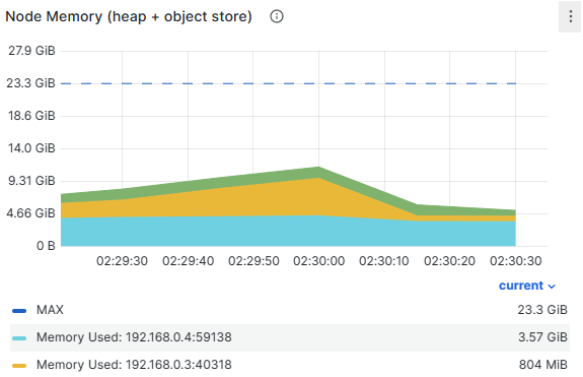


Fig. 7: Sort Memory Usage on Grafana. It can be observed that for the duration of the sort, the memory usage of worker-2 (yellow) keeps increasing until an OOM (out-of-memory) error occurs, while worker-3's memory usage (green) does not increase almost at all, and so does the memory of worker-1 (which is high from the start because of the background processes running).

***Aggregation:*** As expected, Spark performs way better on the aggregate operation, sometimes 10 times as much. This is shown in figures 8 and 9.

***Transformation:*** Same with the load and aggregation tasks, Spark performs better than Ray in our transform script, as shown in table VIII.

***Some Comments on Memory Usage:*** Upon the comparison of memory consumption between Spark and Ray, Ray exhibits higher peak heap memory usage across all benchmarks. For instance, as shown in table IX, Ray's Peak Heap Memory usage for the Load Script is at 1799 MB compared to Spark's which is just at 225 MB. This observed difference in peak heap memory usage suggests that Ray tends to utilize more memory resources than Spark during the execution of ETL tasks. The impact on performance was evident too; As shown in the experiment results, Ray needed more time to execute most ETL tasks. Additionally, the event of disk spilling - which harms performance- was much more prominent in the Ray experiments, and in our experience, Ray was much more prone to OoM (Out-of-Memory) errors. Lastly, upon further investigation by experimenting with limiting the number of usable CPUs in the 3-node Ray cluster (instead of limiting the number of workers), **we did not observe a substantial drop in performance, indicating that for many tasks in Ray, the main bottleneck was the available memory instead of the CPUs.**

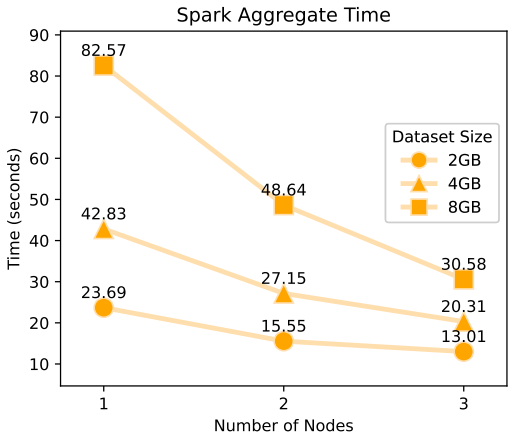


Fig. 8: Spark Aggregation Times

### *C. ML Tasks and Pipelines - Preprocessing and Training of ML models*

***K-Means:*** We will start this subsection by presenting our results for the execution of the K-Means clustering algorithm by following the same transformations in a Ray Dataset and in a Spark Dataframe, as well as by using Spark's MLLib

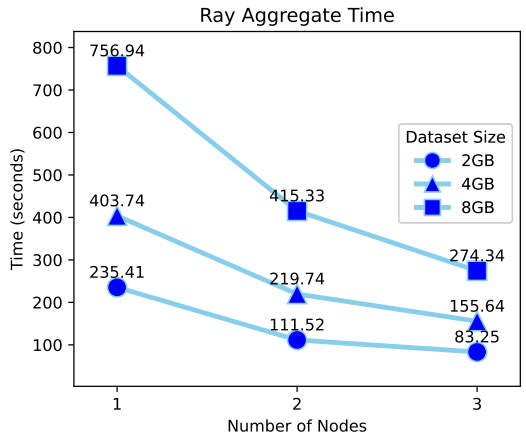


Fig. 9: Ray Aggregation Times

TABLE VIII: Transformation Benchmark

| | | Spark | | Ray | |
|---|---|---|---|---|---|
| **Workers** | **Dataset (GB)** | **Wall** | **CPU** | **Wall** | **CPU** |
| 3 | 2 | 38.12 | 234.55 | 28.73 | 238.71 |
| 2 | 2 | 45.74 | 221.23 | 52.25 | 233.63 |
| 1 | 2 | 66.28 | 212.58 | 85.81 | 229.6 |
| 3 | 4.167 | 56.84 | 420.76 | 66.27 | 486.63 |
| 2 | 4.167 | 85.34 | 412.26 | 92.48 | 477 |
| 1 | 4.167 | 125.37 | 400.52 | 155.56 | 475.06 |
| 3 | 8.333 | 93.96 | 780.46 | 121.26 | 978.84 |
| 2 | 8.333 | 124.26 | 780.23 | 169.19 | 961.36 |
| 1 | 8.333 | 239.41 | 806.95 | 306.33 | 939.2 |

implementation of K-means. All for 10 iterations of the algorithm. These results can be seen in figure 10, where it is... painfully evident that Ray is not appropriate for these kinds of tasks (i.e. intensive data operations on big datasets), since it does not scale well enough for the performance to be considered satisfactory. Even for 10.000-40.000 rows it takes several minutes to run (for reference, Spark's MLLib implementation of Kmeans completes in about 276 seconds for 20 million points of data, amounting to 1.4GBs worth of our dataset).

***Multilayer Perceptron Classifier:*** For our initial training script, we train a Multilayer Perceptron Classifier with one hidden layer on our datasets in table III. Figure 11 shows both frameworks' processing and training times in comparison on the 2GB dataset for different-sized clusters. It is evident that while Spark is faster in the data prepossessing portion, Ray's Trainer dominates in the training part of the task. Then in figure 12, we present the scripts' ability to **scale** on the three datasets (2, 4, 8 GB) across the 3 node cluster. Collective data from our MLP classification benchmark are presented in table

TABLE IX: Peak Heap Memory Used on ETL Tasks (3 Nodes)

| | | Spark | Ray |
|---|---|---|---|
| **Script** | **Dataset** | **Peak Heap Memory** | **Peak Heap Memory** |
| Load | 8.333 | 225 | 1799 |
| Aggregate | 8.333 | 349 | 3794 |
| Aggregate | 4.167 | 286 | 2741 |
| Sort | 4.167 | 228 | 2622 |

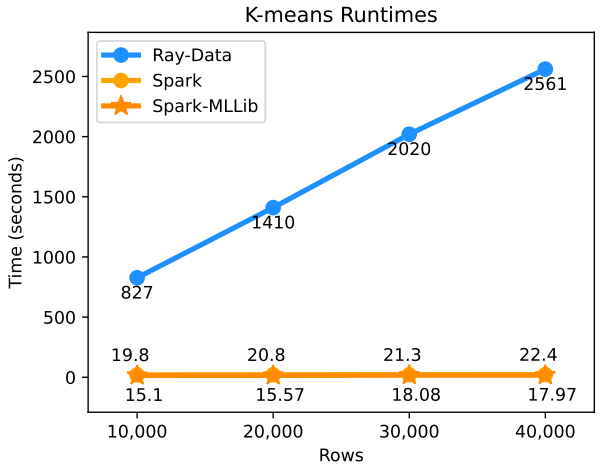


Fig. 10: K-Means Execution Times

X. Overall, our main observation is that the Ray training script, a CPU-intensive task, scores immensely better than Spark.

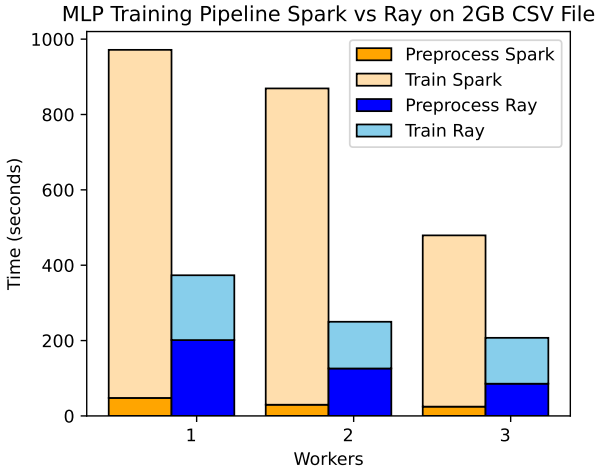


Fig. 11: MLP Training Times

***Hyperparameter Tuning:*** We continue by showing the results of hyperparameter tuning with Spark and Ray on our initial MLP training script. Due to the script's execution time being extremely long, it was only tested on the 4GB CSV file across the scalable cluster, as shown in figure 13. Additionally, as shown in table XI, when using just the head node for execution (1 node cluster), our Spark Tuning method lasted almost 4 hours, while the Ray Tuning method was executed in just 15 minutes, establishing an over 15-time improvement in total runtime, and thus confirming Ray's superiority in scalable ML model training and tuning.

***Random Forest:*** Continuing with our training pipelines, we compared Ray's Random Forest implementation based on XGBoost (see [2]) with Spark's `RandomForestClassifier`. Once again we show that Ray's methods overpower Spark's in terms of training time, while Spark's preprocessing times remain faster. In figure 14 we show the preprocessing and

TABLE X: Collective Classification Timing Results

| Workers | Dataset (GB) | Spark Preprocessing | Spark Train | Spark Total Time | Ray Preprocessing | Ray Training | Ray Total Time |
|---|---|---|---|---|---|---|---|
| 3 | 2 | 24.67 | 454.52 | 479.19 | 85.53 | 121.76 | 207.29 |
| 2 | 2 | 29.74 | 839.45 | 869.19 | 125.91 | 124.00 | 249.91 |
| 1 | 2 | 47.76 | 923.83 | 971.59 | 201.30 | 172.02 | 373.32 |
| 3 | 4.167 | 36.41 | 602.74 | 639.15 | 164.22 | 220.95 | 385.17 |
| 2 | 4.167 | 47.36 | 927.66 | 975.02 | 238.68 | 251.74 | 490.42 |
| 1 | 4.167 | 80.02 | 1555.12 | 1635.14 | 286.08 | 400.01 | 686.09 |
| 3 | 8.333 | 58.95 | 1100.04 | 1158.99 | 305.14 | 410.56 | 715.70 |
| 2 | 8.333 | 89.11 | 1638.05 | 1727.16 | 455.67 | 461.97 | 917.64 |
| 1 | 8.333 | 165.48 | 3466.04 | 3631.52 | 764.91 | 645.78 | 1410.69 |

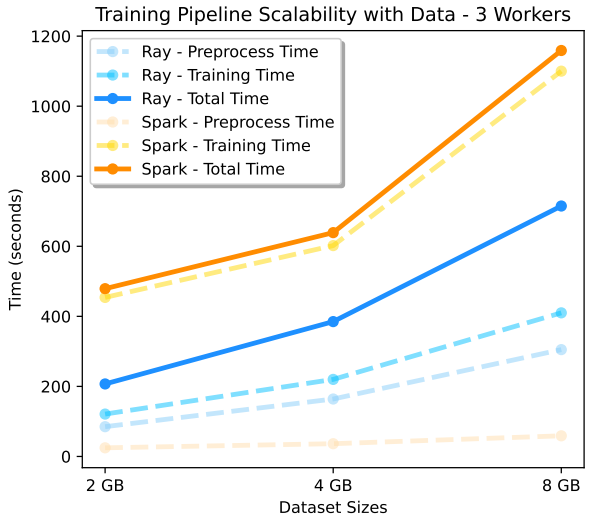


Fig. 12: Ray MLP Training Times

TABLE XI: Hyperparameter Tuning

| Workers | Dataset(GB) | Spark Preprocess | Spark Tune | Ray Preprocess | Ray Tune |
|---|---|---|---|---|---|
| 3 | 4.167 | 38.11 | 7708.77 | 167.47 | 874.07 |
| 2 | 4.167 | 46.62 | 7250.22 | 208.8 | 725.75 |
| 1 | 4.167 | 82.14 | 13740.25 | 355.44 | 850.02 |

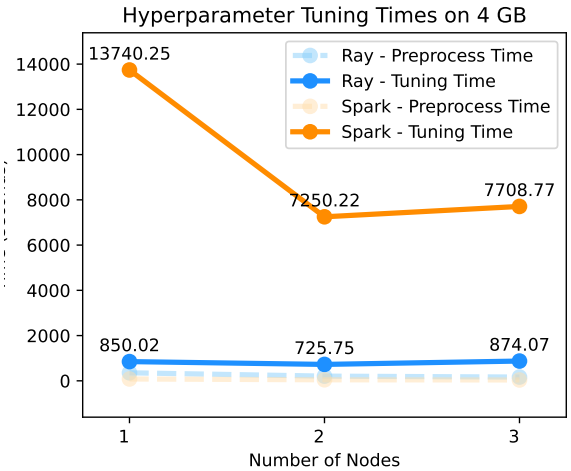


Fig. 13: Hyperparameter Tuning Times (**Note**: For Ray's execution with three workers, more time was needed because at first the 3 out of 4 models were being tested concurrently, and the fourth one only ran with 1 worker, needing more time. For reference, when only three models were tested the runtime (mean across 3 tests) dropped to 652.23s). We suppose that the usage of a scheduler from Ray Tune could alleviate this phenomenon.

training times of both scripts, where Ray's XGBoost Trainer clearly shines on both the 4GB and the 8GB dataset. In table XII we show the execution times collectively on all three CSV files.

TABLE XII: Random Forest Classifiers - Total Times

| Workers | Dataset (GB) | Spark Total Time | Ray Total Time |
|---|---|---|---|
| 3 | 2 | 353.82 | 98.73 |
| 2 | 2 | 368.48 | 144.66 |
| 1 | 2 | 592.17 | 205.07 |
| 3 | 4.167 | 423.48 | 189.23 |
| 2 | 4.167 | 593.73 | 251.18 |
| 1 | 4.167 | 1094.87 | 406.75 |
| 3 | 8.333 | 880.7 | 346.18 |
| 2 | 8.333 | 1023.52 | 468.34 |
| 1 | 8.333 | 1898.08 | 802.08 |

***"Best" of both worlds: Preprocessing with Spark and model training with Ray***: After studying the metrics of the benchmarks on both frameworks, we conclude that Spark is

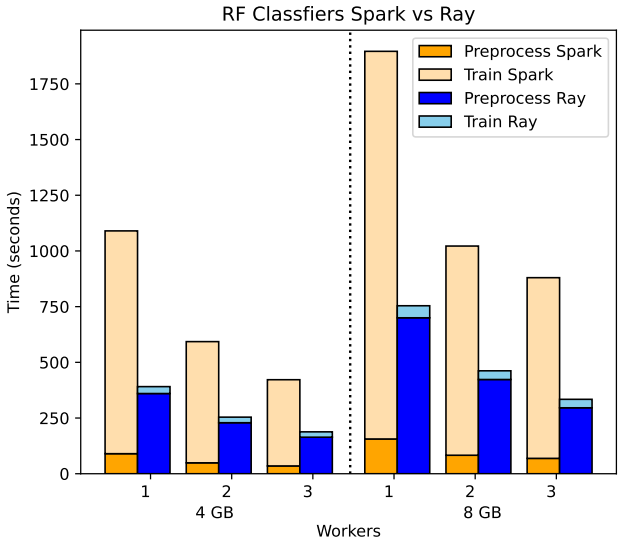


Fig. 14: Random Forest Classifiers - Preprocessing and training times on the 4 GB and 8 GB CSV files, across the cluster.

better suited for intensive data analytics tasks while Ray is much better suited for last-mile preprocessing and distributed machine learning. For this reason, we decided to combine the two frameworks creating a pipeline that uses Spark's methods for preprocessing and Ray's Torch Trainer for training. More specifically, the following actions are executed sequentially:

- The main preprocessing (Transformations) are done with Spark
- The results from the preprocessing are written to the HDFS in parquet form
- Ray reads the preprocessed data in parquet form, and executes last-mile preprocessing (MinMaxScaling, Concatenating features into vectors)
- The MLP model is trained distributedly with a Ray Pytorch Trainer

The results for this pipeline for 3 workers and the 4 GB dataset (in comparison to the previous methods) are presented in figure 15. It seems that the pipeline that combines the best/fastest features of Spark and Ray is faster by a small margin, having additional overhead because of the process of writing back to the HDFS the processed dataset in parquet form and then reading it with Ray. However, if we take into account that the preprocessing task in hand is minimal compared to the typical big data analytics workload, we can infer that the considerable time saved by using Spark for this workload makes this additional overhead more than worth it.

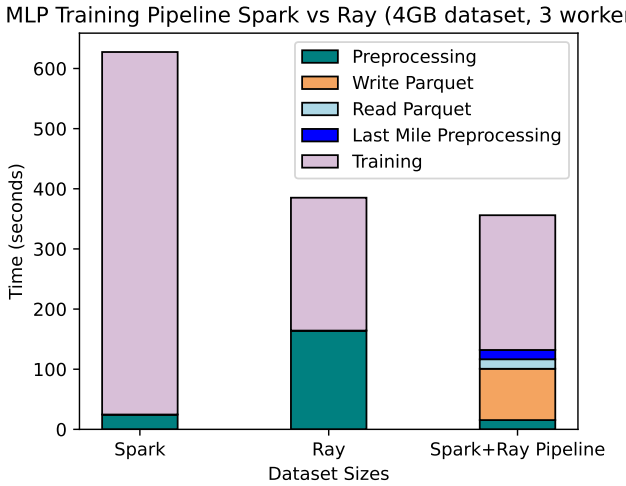


Fig. 15: MLP Training Times

***Feature Extraction From Speech Audio for Emotion Recognition:*** Lastly, we present in figure 16 our results from running the tasks for performing data augmentation and feature extraction on a large dataset of audio files, using Ray's remote functions/Tasks and Spark's UDFs to accelerate the process. We also compare these times to the runtime by simply running the script on a single machine, without Ray or Spark. Ray achieves impressive results by parallelizing this workload with the asynchronous execution of the function for each file in the dataset. Spark, on the other hand, may accelerate the workload to a substantial extent, but it fails to distribute the workload to more workers, even though it is easily parallelizable since it is executed independently for each data row/audio file. It is mentioned in this article [11] that Spark considers User-Defined Functions as black boxes, and thus doesn't perform optimizations like it does with Spark-SQL built-in functions. It was observed that Spark only defined 2 tasks for all the audio files, not allowing the utilization of more workers. This is unfortunate since with Ray this limitation does not exist and any custom piece of code (including independent/repeated usage of external library functions) can be run in parallel with its **Task and Actor low-level APIs**.

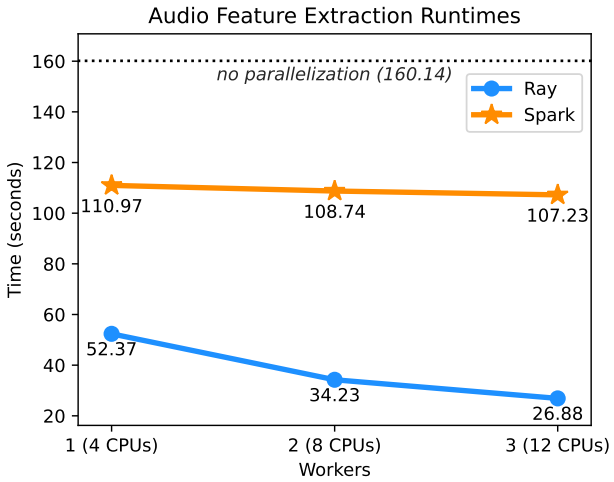


Fig. 16: Audio Feature Extraction Runtimes for Ray and Spark, and comparison with no parallelization

## VI. Conclusion - Discussion

After our extensive experimentation with both Ray and Apache Spark that led to the results presented in the previous section, as well as our general experience with interacting with these systems, our findings can be summarized as follows:

- Apache Spark excels at intensive data analytics tasks. This was concluded by the tasks that consisted of applying the same (to the extent that it was possible) operations on a large dataset.
- Ray, on the other hand, shines in tasks centred around distributed ML. This is achieved by the seamless compatibility with powerful ML frameworks like Pytorch, XGBoost, HuggingFace (Transformers), and many others.
- This leads to each system **being able to accentuate each other's power** when they are combined since **they perform impressively in complementary tasks**; Spark can very efficiently perform data analytics on huge amounts of data to extract useful information and Ray can perform distributed Machine Learning using that information (example: Using Spark to calculate very quickly the number of triangles each node participates in within a graph, PageRank scores, etc., and then using this information to efficiently train an ML model with

Ray that performs, for instance, link prediction for the non-existent edges in that graph).

- Ray is quite memory-hungry, especially when compared to Spark. This, along with the fact that only 30% of each machine's available RAM was allocated to Ray's common Object Store, led to disk spilling being a common effect, that harmed performance in some cases, most frequently related to data analytics tasks. OoM errors were also common, and this led to needing to further examine and monitor the tasks' executions.
- Ray makes parallelizing code seamless with its Actor and Task APIs. In our experience, there were several cases where it was possible to parallelize Python code by only changing a couple of lines, most commonly related to adding the '@ray.remote' decorator to Python functions. So in contrast with Spark, it was possible to run Python code distributedly with minimal refactoring. This however sometimes came with the cost of not always taking advantage of the optimizations that Spark's libraries provided. This brings us to our next point:
- Spark has the privilege of being a more mature ecosystem, which leads to having many more available libraries with high-level APIs which offer optimized execution for various data analytics and ML tasks. Ray, however, is still rapidly growing, and while some features may not be as mature or experimental, it has shown great potential in handling the tasks we are examining.
- Spark's immense power in data analytics operations comes from the internal optimizations that can be utilized through its plethora of high-level APIs. This comes in contrast to Ray's philosophy, which provides universal APIs (Tasks and Actors) that can be leveraged to parallelize any code with potentially impressive results. This leads to each system demonstrating impressive results in different use case scenarios: There are tasks which have distributed algorithms that are difficult to manually implement -efficiently- (like PageRank on a graph), and Spark's many libraries with optimized implementations trivialize tackling these tasks efficiently. Spark can also perform extremely fast and fault-tolerant transformations and queries on huge datasets with Spark-SQL built-in functions. In contrast, for tasks that can be easily parallelized (e.g. 'for/foreach' loops) but do not consist of actions for which optimized Spark implementations exist (e.g. external libraries like librosa), Ray shows great potential.

In conclusion, our comparative study has led to a deeper understanding of how Ray and Apache Spark perform in real-world big data projects. By evaluating their efficiency and scalability across different workload sizes and available resources, we have identified their respective roles as powerhouses in different, but not necessarily mutually exclusive aspects of Big Data Workloads. Our findings reveal that both Ray and Apache Spark exhibit impressive capabilities in different aspects of data analytics tasks and Machine Learning and Artificial Intelligence pipelines, making them powerful tools for data-intensive operations. The choice between Ray and Apache Spark may depend on the specific requirements and priorities of a given project, but their usage is not mutually exclusive, and having a deep knowledge of their strengths and weaknesses can greatly contribute to a given project's scalability and success. Our comparative study demonstrates that Ray and Apache Spark are not strictly competitors, but rather complementary frameworks that can enable data scientists and engineers to tackle challenging and complex problems in the era of Big Data and AI.

## REFERENCES


[1] KONECT – the koblenz network collection. 1. Accessed: 2024-01-25.

[2] Random forests(tm) in xgboost. https://xgboost.readthedocs.io/en/stable/tutorials/rf.html.

[3] Adhitya Bhawiyuga and Annisa Puspa Kirana. Implementation of page rank algorithm in hadoop mapreduce framework. In *2016 International Seminar on Intelligent Technology and Its Applications (ISITIA)*, pages 231–236, 2016.

[4] Shivam Burnwal. Speech emotion recognition python notebook. *Kaggle*, 2022.

[5] Luca Canali. sparkmeasure: Performance troubleshooting tool for apache spark. https://github.com/LucaCanali/sparkMeasure, 2023.

[6] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E. Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training. 7 2018.

[7] Steven R. Livingstone and Frank A. Russo. Ravdess emotional speech audio, 2019.

[8] Brian McFee et al. librosa: Audio and music signal analysis in python - v. 0.10.1 docs. https://librosa.org/doc/latest/index.html.

[9] Brian McFee, Colin Raffel, Dawen Liang, Daniel PW Ellis, Matt McVicar, Eric Battenberg, and Oriol Nieto. librosa: Audio and music signal analysis in python. In *Proceedings of the 14th python in science conference*, pages 18–25, 2015.

[10] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging ai applications, 2018.

[11] Naveen (NNK). Pyspark udf (user defined function) - spark by Examples. https://sparkbyexamples.com/pyspark/pyspark-udf-user-defined-function/, 2023.

[12] Max Pumperla, Edward Oakes, Richard Liaw, Boston Farnham, Sebastopol Tokyo, Beijing Boston, Farnham Sebastopol, and Tokyo Beijing. *Learning Ray Flexible Distributed Python for Data Science*. 2023.

[13] Ray. Dataset.stats — ray 2.9.1. https://docs.ray.io/en/latest/data/api/doc/ray.data.Dataset.stats.html, 2024.

[14] Ray Development Team. Configuring and managing ray dashboard. https://docs.ray.io/en/latest/cluster/configure-manage-dashboard.html#observability-visualization-setup. Accessed on 20 Jan 2024.

[15] Salman Salloum, Ruslan Dautov, Xiaojun Chen, Patrick Xiaogang Peng, and Joshua Zhexue Huang. Big data analytics on apache spark, 11 2016.

[16] Ray Team. Advanced: Performance tips and tuning. https://docs.ray.io/en/latest/data/performance-tips.html.

[17] Ray Team. Cluster key concepts. https://docs.ray.io/en/latest/cluster/key-concepts.html. Accessed on 16 Jan 2024.

[18] Ray Team. Key concepts of ray tune. https://docs.ray.io/en/latest/tune/key-concepts.html.

[19] Joel Tok. Memory optimisation – python dataframes vs lists and dictionaries (json-like). https://www.joeltok.com/posts/2021-06-memory-dataframes-vs-json-like/, 2021.

[20] Xinyu Xing. An anatomy of the implementation of the raft consensus algorithm. https://knowledge-shareing-xinyu.blogspot.com/2016/03/an-anatomy-of-implementation-of.html, 2016. Accessed on 20 Jan 2024.