Benchmarking Ray vs Apache Spark: A comparative Analysis on Distributed ETL, Graph Processing and Machine Learning Workloads

Στέφανος Θέος (AM: 03119219)
School of Electrical and Computer Engineering
National Technical University of Athens
Athens, Greece
el19219@mail.ntua.gr / Group 27

Αθανάσιος Μπιτσάνης (AM: 03119136)
School of Electrical and Computer Engineering
National Technical University of Athens
Athens, Greece
el19136@mail.ntua.gr / Group 27

***Abstract*— As the demand for processing large-scale datasets continues to grow, efficient distributed computing frameworks are essential for modern data-intensive applications. This project presents a comparative evaluation of Ray and Apache Spark, focusing on their performance across a variety of real-world workloads including ETL operations, graph algorithms (e.g., PageRank and triangle counting), and machine learning pipelines. Experiments are executed on a multi-node cluster with HDFS-backed storage, using both synthetic and real datasets. Key metrics such as execution time, CPU utilization, memory footprint, and scalability across workers are measured. The results show that Apache Spark excels in batch data transformations and large-scale joins, while Ray demonstrates strengths in parallelizable and dynamic ML tasks due to its lightweight task scheduling. This analysis aims to guide practitioners in choosing the most suitable framework depending on the computational workload and resource constraints.**

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

AVAILABILITY

LINK TO GIT

LINK TO SPREADSHEET

I. Introduction

In the modern age of digital transformation, data has emerged as one of the most valuable and pervasive assets across industries. Whether generated from user interactions on social media, sensor readings from IoT devices, transactional records in e-commerce, or logs from software systems, data is produced at an unprecedented rate. This exponential growth of data has given rise to the term "big data," denoting datasets so large and complex that traditional data processing methods are no longer sufficient to handle them effectively.

As organizations and researchers increasingly rely on data-driven insights, the ability to process and analyze vast volumes of data has become a critical priority. Efficient data handling is not merely a matter of convenience—it directly impacts decision-making, system performance, and the feasibility of deploying real-time, scalable applications. Consequently, there is a growing demand for robust frameworks that can manage, process, and extract value from massive datasets using distributed computing resources.

To meet this demand, a number of scalable data processing frameworks have been developed, particularly within the Python ecosystem, which remains a cornerstone of the data science and machine learning communities. Among these, Apache Spark and Ray have garnered significant attention for their ability to distribute computations across multiple nodes and efficiently handle both structured and unstructured data. Each of these frameworks is equipped with unique architectural features and performance characteristics, making them suitable for different use cases and workloads.

This report investigates and compares the performance of Ray and Apache Spark through a series of experiments conducted in a distributed cluster environment. The experiments span a variety of tasks commonly encountered in large-scale data processing pipelines, including ETL (Extract, Transform, Load) operations, graph analytics such as PageRank and Triangle Counting, clustering with K-Means, and machine learning workflows using models like MLP and XGBoost. The datasets used include both synthetically generated and real-world data (such as those from the ASHRAE Energy Prediction challenge), processed at scale through Hadoop's HDFS.

The objective of this project is to evaluate the strengths and limitations of each framework with respect to execution time, CPU usage, memory efficiency, and scalability. Rather than focusing on the algorithmic accuracy of the models employed, the emphasis lies in assessing the practicality of Ray and Spark for high-performance, distributed data processing. This comparative study aims to provide data engineers and practitioners with actionable insights into selecting the appropriate tool for various large-scale analytics workloads.

II. Framework Review and Experimental Environment

A. Okeanos-Knossos [1]

Okeanos-Knossos is a cloud computing platform developed and maintained by the Greek Research and Technology Network (GRNET). It is designed to offer Infrastructure-as-a-Service (IaaS) to the Greek academic and research community, enabling users to deploy and manage virtual machines (VMs), private networks, and storage resources with ease. Built on top of OpenStack technologies, Okeanos-Knossos provides a reliable and flexible environment for hosting distributed applications, running

large-scale data processing tasks, and performing computational experiments. Its user-friendly interface, API support, and integration with federated authentication services make it a powerful tool for researchers and students who require scalable resources without the need to maintain physical hardware. In this project, Okeanos-Knossos served as the backbone infrastructure for deploying a cluster of VMs, each configured to support distributed execution of data processing frameworks such as Ray and Apache Spark.

B. HDFS and Hadoop YARN[2],[3]

Hadoop Distributed File System (HDFS) and Hadoop YARN are core components of the Apache Hadoop ecosystem, widely used for scalable and fault-tolerant distributed computing. HDFS is a highly reliable file storage system designed to store large volumes of data across multiple machines in a cluster. It splits files into blocks and distributes them across nodes, ensuring redundancy and high availability. On the other hand, YARN functions as the cluster's resource manager, allocating CPU and memory resources to various applications running in parallel. It efficiently handles scheduling and job execution, making it an ideal choice for managing workloads in big data environments. In this project, HDFS was used to store the datasets and graph structures processed by the Ray and Spark frameworks, while Hadoop YARN served as the execution backend for Spark, allowing for distributed processing across multiple nodes within the Okeanos-Knossos infrastructure. Together, these components provided a robust foundation for evaluating the performance of large-scale data processing tasks.

C. Ray[4]

Ray is an open-source distributed computing framework designed to scale Python-based applications seamlessly across multiple cores and nodes. It provides a flexible and easy-to-use interface for parallel and distributed execution, making it well-suited for a wide range of workloads including data processing, machine learning, hyperparameter tuning, and reinforcement learning.

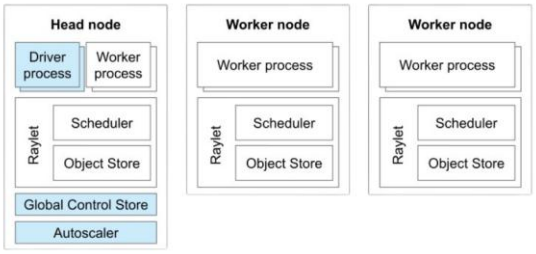

Fig. 1. Ray Cluster Overview

Unlike traditional big data tools, Ray is built with a focus on Python developers and supports dynamic task graphs and fine-grained parallelism, offering high performance without requiring complex configurations. In this project, Ray was utilized to implement and evaluate large-scale data operations such as ETL, machine learning model training, and graph processing (e.g., PageRank and Triangle Counting). By leveraging Ray's built-in Dataset API and remote task execution capabilities, we were able to distribute computations efficiently across a cluster of virtual machines, allowing us to assess Ray's scalability and responsiveness in comparison to Apache Spark.

D. Spark[5]

Apache Spark is a powerful, open-source distributed computing system widely used for big data processing and analytics. Designed for speed and scalability, Spark enables in-memory data computation and supports a wide range of operations including batch processing, SQL queries, machine learning, and graph analytics through its modular components. At the heart of Spark lies the concept of Resilient Distributed Datasets (RDDs), an abstraction that represents fault-tolerant, distributed collections of objects that can be operated on in parallel. RDDs allow for fine-grained control over data operations and are foundational to Spark's ability to process large datasets efficiently.

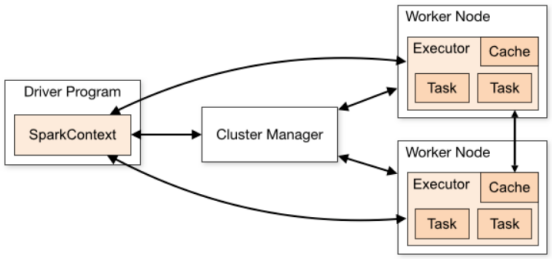

Fig. 2. Spark Cluster Overview

It operates on top of the Hadoop ecosystem and is highly compatible with the Hadoop Distributed File System (HDFS) and Hadoop YARN for cluster resource management. Spark's DataFrame API and support for parallel execution make it particularly suitable for complex data transformation workflows and iterative machine learning algorithms. In this project, Spark served as a benchmark framework against which Ray was compared. Spark was deployed across multiple virtual machines in a distributed environment, where we used it to perform ETL tasks, train machine learning models, and run graph algorithms, allowing us to systematically evaluate its performance and resource efficiency relative to Ray.

III. Environment Setup and Configuration

A. Virtual Machines

For the purposes of this project, we were assigned three virtual machines (VMs) provisioned by Okeanos, GRNET's cloud service, which shared identical system specifications:

- OS: Ubuntu Server LTS 22.04
- CPU: 4 virtual CPUs
- Memory: 8 GB RAM
- Storage: 30 GB disk capacity
- Python 3.10.12

The master node was assigned a public IP address to allow external access to cluster services, while all nodes were interconnected via a private internal network for inter-node communication. Secure access to each VM was achieved

through SSH, with port forwarding (tunneling) configured to expose only necessary services in a controlled manner.

B. Apache Spark and Hadoop Configuration

The installation and configuration of Hadoop and Spark followed the guidelines provided in the [Hadoop Advanced DB course material]. This process primarily involved setting up the runtime environments for both systems across the cluster. Once the environment was configured, initializing the HDFS and YARN resource manager required only the execution of start-dfs.sh and start-yarn.sh on the master node. To enable Python-based development, PySpark was installed to support running Spark applications with Python. Additionally, the Spark History Server was configured and launched using the $SPARK_HOME/sbin/start-history-server.sh script, allowing access to the execution history of completed jobs through a web interface.

C. Ray Setup and Configuration

To set up the Ray environment, we first created a Python virtual environment on each of the three virtual machines and installed the required dependencies using 'pip install ray[data,train,tune]', along with all additional packages specified in the requirements.txt file of our GitHub repository. This ensured compatibility with the various Ray modules used throughout our project, including Ray Core, Data, Train, and Tune.

The Ray cluster was composed of a single head node and two worker nodes. The head node not only hosted a worker process but was also responsible for managing cluster coordination through its control components, such as the driver process, global control store, and autoscaler. We initiated the cluster from the head node using the following command:

```
ray start --head --node-ip-address=192.168.0.4 \
--port=6379 --object-store-memory=2147483648 \
--dashboard-host=0.0.0.0 \
--system-config='{"automatic_object_spilling_enabled": true, "object_spilling_threshold": 0.8}'
```

This setup limited the object store memory to 2GB and enabled object spilling to disk once memory utilization exceeded 80%, mitigating Ray's relatively high memory demands. The worker nodes were then connected to the cluster using:

```
ray start --address='192.168.0.4:6379'
```

In both commands, we included the HDFS CLASSPATH environment variable, allowing Ray processes to interface with the distributed file system, as all nodes were configured to read input data directly from HDFS.

D. NFS Distributed Storage for Checkpoints/Logs [6]

To support distributed training and centralized storage of checkpoints and logs, a Network File System (NFS) server was configured on the cluster master node by installing nfs-kernel-server. A shared directory (/mnt/shared) was created, granted open permissions, and exported to the worker nodes using /etc/exports, by adding in the following line:

```
/mnt/shared 192.168.0.4(rw,sync,no_subtree_check) 192.168.0.6(rw,sync,no_subtree_check)
```

The NFS service was then restarted using:

```
sudo systemctl restart nfs-kernel-server
```

Worker nodes mounted the shared directory permanently by installing nfs-common and appending the following entry to their /etc/fstab:

```
okeanos-master:/mnt/shared /mnt/shared nfs defaults 0 0
```

This setup allowed Ray's XGBoostTrainer to write results and logs to a common location accessible by all nodes, which is essential for coordinated tuning, monitoring, and reproducibility in distributed machine learning workflows.

IV. DATA

To support large-scale benchmarking and experimentation, we created a Python script (data_generator.py) that produces synthetic tabular data with both numerical and categorical features. Numeric data is generated using make_classification from sklearn.datasets, while categorical fields include random integers and synthetic text via the Faker library. The data is streamed directly in CSV format without local disk I/O. A shell script (upload_data_to_hdfs.sh) pipes this stream into HDFS using hdfs dfs -put, minimizing I/O overhead. We generated datasets of 2, 4, and 8 GBs, simulating realistic classification tasks, and balanced their distribution across datanodes using the hdfs balancer command.

For graph processing benchmarks, we developed graph_generator.py, a NetworkX-based script that generates edge-list TSV files representing synthetic graphs with configurable size and topology. It supports Small-World (Watts–Strogatz), Scale-Free (Barabási–Albert), and Random (Erdős–Rényi) models. These graphs are uploaded to HDFS using upload_graph_to_hdfs.sh, allowing seamless integration into distributed workflows. We also sourced real graphs through the Stanford SNAP project [7]. Here is a brief summary of our files:

- Scale-Free graphs, generated using the Barabási–Albert model. These simulate networks with power-law degree distributions, such as social or citation networks. Each node connects to 5 existing nodes on average, and the graphs are generated with sizes up to 100,000 nodes.
- Small-World graphs, generated using the Watts–Strogatz model. These reflect the structure of many real-world networks, balancing high clustering with short path lengths. The generated graphs use 100 neighbors per node and a rewiring probability of 0.1, with sizes ranging up to 100,000 nodes.
- Random graphs, generated using the Erdős–Rényi model. These graphs use a fixed edge creation probability of 0.01

and are useful as a neutral baseline for topological analysis and algorithm performance comparison.

- 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.

TABLE I. DATASETS USED FOR BENCHMARKING

Rows	#of CSV files	Total Size
25 million	1	2.2 GB
50 million	1	4.4 GB
100 million	1	8.8 GB

TABLE II. DATASET SCHEMA

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

TABLE III. GRAPHS USED FOR BENCHMARKING

Graph	nodes (n)	edges (m)
SW-x	x	100*x
SF-x	x	5*x
Random	x	$0.01*x^2$
higgs	456,626	14,855,842

V. EXPERIMENTAL TASKS AND WORKLOADS

In this section, we describe the set of experimental tasks and workloads used to evaluate and compare the performance of Ray and Apache Spark across different computational paradigms. These tasks were carefully selected to represent a diverse mix of data processing, machine learning, and graph analytics scenarios. Our goal was to assess how each framework handles varying workload characteristics under a consistent cluster configuration.

A. ETL Operations

To evaluate the performance of Ray and Spark in realistic workloads, we implemented a suite of ETL operations over large, synthetically generated datasets stored in HDFS. These included numerical and categorical features as well as classification labels. Each operation—load, join, aggregate, sort and transform—was implemented using both frameworks with equivalent logic to ensure a fair comparison. Below is a concise comparison of the approaches used.

The tasks performed are as follows:

- Load: Both systems load data from HDFS using their respective APIs. In Spark, CSV files are read into DataFrames using read.format("csv"), followed by a triggering action like count() to materialize the dataset. Ray reads the same data using read_csv() and materializes it with materialize() to compute basic statistics via .stats()
- Join: Spark performs native distributed joins between DataFrames by identifying a common key and invoking join(...) on the DataFrames, with execution tracked by SparkMeasure. Ray reads both datasets as Ray Datasets, converts the smaller one to a pandas DataFrame, and broadcasts it. It then applies a map_batches function to perform the join on each partition, allowing efficient execution despite the lack of built-in shuffle join support.
- Aggregate: Spark uses the DataFrame API to group by categorical_feature_2 and aggregate the sum of feature_4, optimized by the Catalyst engine. Ray performs a similar operation using groupby(...).sum(...), with manual control over batching for parallelism but without Spark's optimizer.
- Sort: Spark applies a sort() operation on the selected column (feature_1) with automatic query planning. Ray implements sorting using the Dataset API, leveraging parallelism but sometimes incurring higher overhead due to memory pressure on large datasets.
- Transform: Spark performs transformations by adding a new column (new_feature = sqrt($f1^2 + f2^2$)) and applying a row-level filter based on string length. Ray uses StandardScaler to normalize numeric features and applies map() and filter() functions for the same logic, with all operations running in parallel over batches.

For these tasks, we used a synthetic dataset stored as multiple CSVs in HDFS. The files are loaded and converted into Spark DataFrames and Ray Datasets respectively. We conducted experiments on datasets ranging from small to sizes that no longer fit into a single machine's main memory. We also varied the number of workers to observe how each system scales with increased computational resources and memory availability.

B. Graph Operations

Pagerank: PageRank is an iterative graph algorithm that estimates the importance of each node based on the structure of incoming links, originally developed for ranking web pages. It works by distributing "rank" scores through the network and updating them across multiple iterations based on link structure. We evaluate its performance on both Spark and Ray using a directed graph loaded from HDFS. In Spark, we rely on the built-in pageRank() method from the GraphFrames library, which handles iterations internally. In Ray, we manually implement the algorithm by simulating message passing across nodes and recomputing ranks in parallel tasks over multiple iterations. Although Ray provides more control over the algorithm's internals, Spark benefits from built-in optimization and abstraction. We run both implementations on the same input and track runtime, memory usage, and CPU consumption to compare their ability to scale across larger graphs and more workers.

Triangle Counting: Triangle Counting is a classic graph analytics task used to detect tightly connected communities within a network. It counts how many triangles (three-node closed loops) exist in an undirected graph. For our evaluation, both Spark and Ray load the graph structure from HDFS and compute the total number of triangles. Spark uses the built-in triangleCount() method from the GraphFrames library, which abstracts the process and efficiently handles the computation on distributed data. Ray implements the algorithm explicitly by distributing the adjacency list across tasks and counting triangles in parallel per node chunk. This approach offers flexibility but requires more manual handling. We measure

execution time, CPU, and memory usage across systems to understand how each handles the computational load as the graph scales in size and the number of workers increases.

C. ML Algorithms and Pipelines

In order to thoroughly assess the performance of Ray and Apache Spark in real-world scenarios, we focus on a set of representative Machine Learning tasks. Apache Spark provides MLlib, a robust library for scalable machine learning that supports a wide range of algorithms and transformations in a distributed fashion. Ray, in contrast, offers seamless integration with modern machine learning frameworks such as PyTorch, TensorFlow, and XGBoost, enabling distributed training and fine-tuning across large datasets through Ray Datasets. By leveraging each system's native capabilities, we are able to benchmark their efficiency, scalability, and ease of use across several ML workflows.

K-means Clustering: This is a fundamental unsupervised learning algorithm used to partition a dataset into k distinct clusters. It operates by iteratively assigning each data point to the nearest centroid, then updating those centroids based on the mean of the assigned points. In graph analysis, this technique can be applied to node degree—a basic structural property representing how many connections each node has—to uncover groups of nodes with similar levels of connectivity.

In this project, K-Means clustering was applied to the degree distribution of large graphs. Using Apache Spark, node degrees were computed and transformed into feature vectors with VectorAssembler, followed by distributed clustering using Spark MLlib's KMeans. For Ray, the graph was partitioned across multiple workers, degrees were computed in parallel, and clustering was performed using Scikit-learn's KMeans after aggregating the results.

The task allowed us to evaluate how each framework handles parallel processing and numeric-heavy workloads. We recorded metrics such as execution time, CPU utilization, and memory usage to compare performance across different cluster sizes and numbers of clusters (k).

More Complex ML Workloads:

To evaluate the capabilities of Ray and Apache Spark under more demanding machine learning workloads, we designed a task centered on hyperparameter tuning for supervised classification using three commonly employed models: a simple Neural Network (MLP), XGBoost, and a Random Forest Classifier. The dataset used includes both numerical and categorical variables and is stored in HDFS. Preprocessing steps are consistent across systems and include computing a derived feature (new_feature), dropping unused columns, and selecting relevant predictors. For Spark, preprocessing is performed using a VectorAssembler followed by MinMaxScaler, while Ray leverages ray.data for transformations before converting the dataset to NumPy arrays for training.

Each framework then executes a distributed hyperparameter search using its respective tools: Spark employs CrossValidator from MLlib with ParamGridBuilder and 3-fold cross-validation, while Ray utilizes ray.tune in conjunction with tune.Tuner and tune.with_parameters to distribute trials across available CPU resources. Each model—MLP, XGBoost, and Random Forest—was implemented in both environments using compatible libraries: pyspark.ml.classification and xgboost4j-spark for Spark, and PyTorch/XGBoost/Scikit-learn for Ray. For all cases, we report both the total execution time and the best model accuracy, offering insights into how well each system handles complex, iterative ML workloads at scale.

Hyperparameter Optimization: Hyperparameter tuning is a computationally intensive process, as it involves repeatedly training a model across different combinations of hyperparameters and evaluating performance. In our approach, we utilize the parallelism capabilities of both Ray and Apache Spark to efficiently conduct hyperparameter optimization for three distinct models: a simple Neural Network (MLP), XGBoost, and a Random Forest Classifier. For the Spark-based experiments, we relied on CrossValidator and ParamGridBuilder from the pyspark.ml.tuning library to perform exhaustive search with 3-fold cross-validation. On the Ray side, we utilized Ray Tune's Tuner interface to distribute training runs across workers, pairing it with tune.with_parameters to cleanly inject data and configuration into each trial. Each tuning script was designed to track the best performing configuration based on validation accuracy, enabling a fair comparison of how each framework scales and manages parallel hyperparameter search workloads.

ASHRAE Energy Prediction Task: For the ASHRAE energy prediction task, both Apache Spark and Ray were employed to construct distributed machine learning pipelines aimed at forecasting hourly building energy consumption. The data, residing in HDFS, combines time-series meter readings with weather and building metadata. After joining the datasets on building_id, timestamp, and site_id, rows with missing values were filtered out and the target variable (meter_reading) was log-transformed using log1p. Categorical variables, such as primary_use, were one-hot encoded, while continuous features were standardized and assembled into feature vectors.

The Ray pipeline leveraged ray.data for distributed preprocessing and employed XGBoostTrainer from ray.train, paired with ray.tune for hyperparameter tuning using a grid search over eta and max_depth. System-level metrics including CPU and memory usage were monitored using psutil, and training duration was recorded manually. On the other hand, the Spark pipeline utilized a GBTRegressor integrated into a full Pipeline along with StringIndexer, OneHotEncoder, and StandardScaler. Model selection was handled via CrossValidator, and performance profiling was performed using sparkmeasure. Additional Spark tuning included caching intermediate DataFrames and configuring execution parameters such as spark.sql.shuffle.partitions and executor memory.

This setup ensured a controlled and consistent environment for benchmarking the two frameworks, offering a direct comparison of their effectiveness and efficiency under a realistic, large-scale ETL and modeling workload. To facilitate shared logging and checkpointing across nodes, an NFS server was mounted at /mnt/shared, accessible to both the master and worker nodes, enabling synchronized storage during distributed training.

VI. BENCHMARKING AND EXPERIMENTAL RESULTS

In this section, we present the benchmarking results of our experiments, comparing the performance of Ray and Apache Spark across various data processing and machine learning tasks. The benchmarks were conducted on a shared virtualized environment using 1, 2, and 3 worker nodes and multiple dataset types and volumes (synthetic, structured CSVs, graph data, and). Key performance metrics evaluated include Runtime, CPU Usage, and Peak Heap Memory Consumption. For metric collection, we leveraged SparkMeasure and YARN for Spark, and Ray's built-in stats() API, as well as custom monitoring via Python's psutil and resource modules for Ray. The Ray Dashboard and Spark History Server were also utilized to validate and supplement our measurements. Each experiment was repeated at least three times to obtain average values, reducing the impact of outliers or resource fluctuations across runs.

Finally, it's important to highlight that the reported metrics represent the average results from three runs of each experiment. This approach ensures greater accuracy and robustness by minimizing the influence of outliers and reducing the effect of run-to-run variability.

A. ETL Operations

Our experiment was conducted using as input the 3 generated datasets of sizes 2.2,4.4 and 8.8 GB and utilizing 1,2 and 3 worker nodes. For each operation we analyze the results of our testing:

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

Sort Operation: In this taks the efficiency of Spark is clearly displayed in the graphs below, with the sorting times being half of the Ray's:

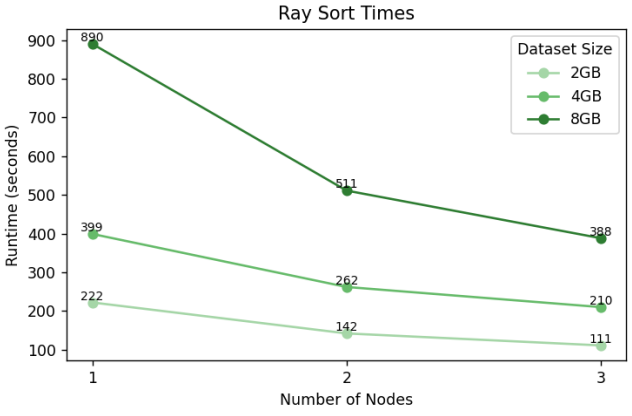

Fig. 3. Ray Sort Runtime

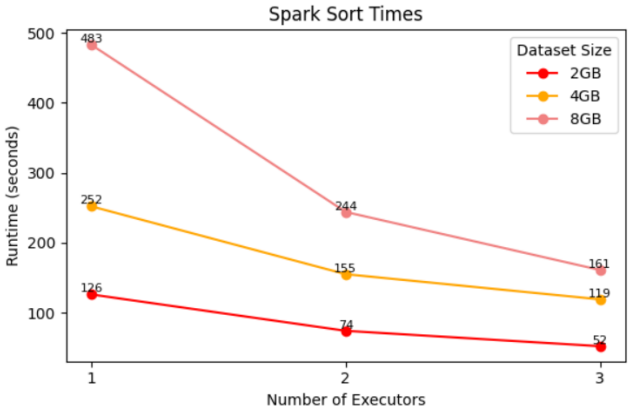

Fig. 4. Spark Sort Runtime

Aggregate Operation: During the aggregation operation it is clear that Spark outperfoms Ray. As shown in figures 5 and 6, Ray's times are 10 times those of Spark's.

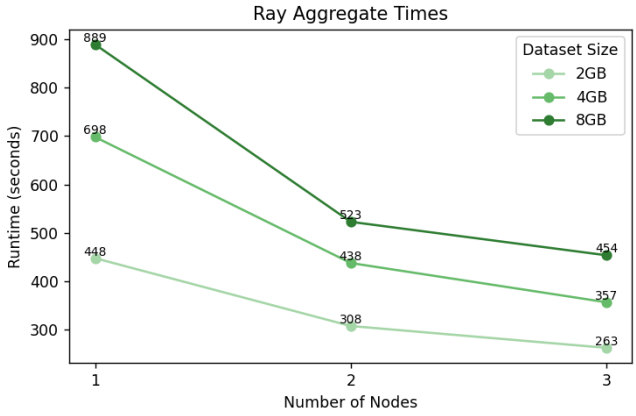

Fig. 5. Ray Aggregate Runtime

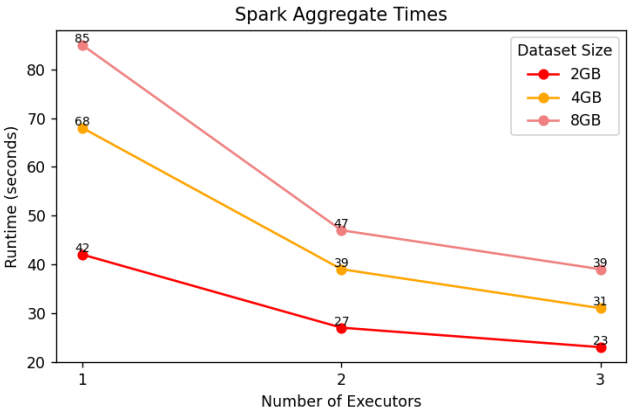

Fig. 6. Spark Aggregate Runtime

TABLE IV. LOADING BENCHMARK RESULTS

		Ray				Spark			
Workers	**Dataset (GB)**	**Runtime (s)**	**CPU (s)**	**Tasks**	**Peak Heap Memory**	**Runtime (s)**	**CPU (s)**	**Tasks**	**Peak Heap Memory**
3	2.2	36,54	38,15	134	2426 MB	55,41	117,1	20	568,93 MB
2	2.2	38,73	36,37	134	2457 MB	67,79	109,3	20	513,93 MB
1	2.2	40,22	36,21	134	2387 MB	121,6	109,7	20	480,53 MB
3	4.4	39,15	55,25	48	2426 MB	84,02	264,4	77	587,63 MB
2	4.4	43,91	61,88	48	2527 MB	124,5	278,5	77	542,79 MB
1	4.4	47,21	58,19	48	2501 MB	155,7	270,2	77	573,25 MB
3	8.8	49,82	129,2	65	2589 MB	120,5	476,3	113	554,3 MB
2	8.8	51,27	138,9	65	2628 MB	138,8	449,2	113	537,3 MB
1	8.8	64,14	128,5	65	2639 MB	186,3	468,9	113	530,7 MB

Transform Operation: This ETL operation follows the same pattern of the previous two and has Spark outperforming Ray. The corresponding results of the transform runtimes can be observed in the figures 7 and 8:

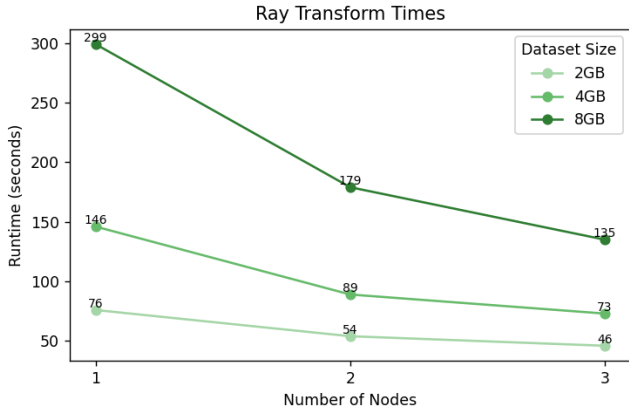

Fig. 7. Ray Tranform Runtime

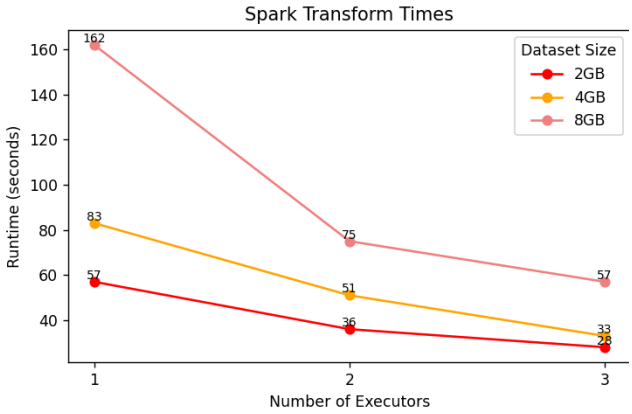

Fig. 8. Spark Tranform Runtime

B. Graph Operations

Pagerank: In our experiments with the PageRank algorithm, we employed synthetic graphs generated using the custom graph_generator.py script, allowing us to systematically evaluate performance across different graph sizes. The Spark implementation leveraged GraphX, an optimized graph processing library that is tightly integrated within the Spark ecosystem, enabling efficient in-memory computation and minimal overhead. In contrast, the Ray implementation relied on a manual construction of the algorithm using Ray Datasets and Python logic. As shown in Figure 9, Spark's runtime scaled almost linearly with increasing graph size, while Ray's execution time grew geometrically, resulting in significant performance disparities on larger datasets. This highlights the maturity and specialization of GraphX for iterative graph computations, whereas Ray—despite its flexibility—lacks dedicated primitives for graph operations, making implementations more cumbersome and less performant.

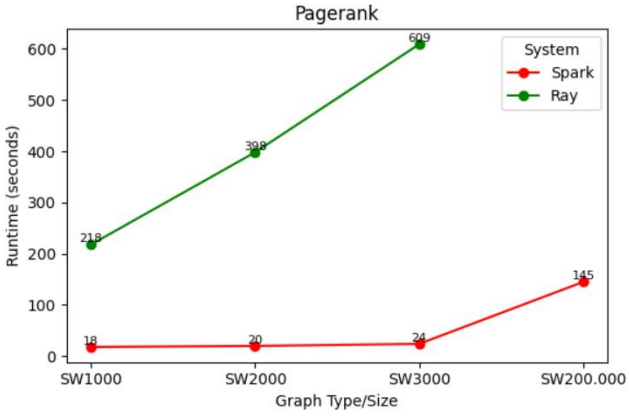

Fig. 9. Pagerank Runtimes

Triangle Counting: Triangle counting in Spark is performed using the GraphFrames library, which builds on Spark's DataFrame API and executes the algorithm through multiple distributed join and aggregation operations. This approach benefits from Spark's scalability but introduces overhead due to the cost of relational operations across partitions.

In contrast, the Ray implementation uses NetworkX directly, constructing the graph in memory and parallelizing the computation by distributing subsets of nodes to Ray workers. Each worker executes a local triangle count using efficient set intersections on adjacency lists, avoiding costly joins.

This fundamental difference explains the performance gap: Ray executes faster on moderate-sized graphs due to reduced coordination overhead, while Spark scales more consistently but requires more time due to its general-purpose

execution model. The performance of the two systems can be seen on the graph below (Fig. 10):

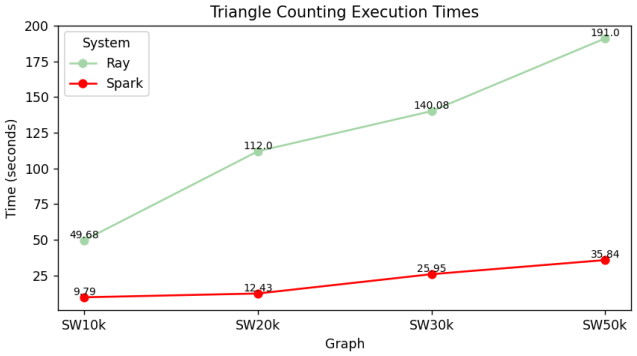

Fig. 10. Triangle Counting Runtimes

C. ML Operations and Pipelines

K-Means clustering: For the execution of the the K-Means clustering algorithm equivalent transformations on a Ray Dataset and a Spark DataFrame were used, along with Spark's optimized MLLib implementation. All experiments were conducted with 10 iterations and 5 clusters. As shown in Figure 11, the results clearly indicate that Ray is not well-suited for this type of task—namely, large-scale data processing and transformation—due to poor scalability and performance. Even with relatively modest datasets (10,000–50,000 rows), execution times extend to several minutes.

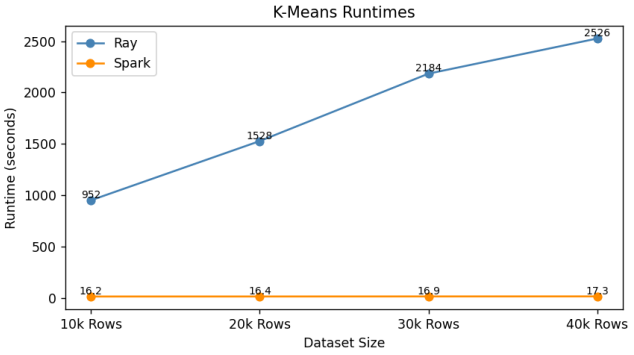

Fig. 11. K-Means Runtimes

Hyperparameter Training and Tuning: For Neural Network hyperparameter tuning, Ray leverages its PyTorch integration and Ray Datasets to enable efficient distributed training with flexible model design. In contrast, Spark uses its MLlib MultilayerPerceptronClassifier within a pipeline. Despite Spark's JVM optimizations, Ray's lightweight Python-native execution and batch processing led to faster runtimes for the NN task, while still achieving competitive AUC scores.

The Random Forest hyperparameter tuning was performed using Ray's distributed training with SklearnTrainer and Ray Datasets for the Python-based pipeline, while Spark leveraged its optimized MLlib pipeline within the JVM environment. Spark's native integration of data processing and Random Forest training, combined with efficient cluster resource management, led to faster runtimes and better scalability. Ray's Python-centric approach, despite its flexibility and convenient tuning APIs, incurred overheads in data serialization and coordination, causing slower performance on large datasets.

We executed the Neural Network and Random Forest training and hyperparameter tuning scripts on a 3-node cluster, using three different dataset sizes—2 GB, 4 GB, and 8 GB. In each run, the scripts trained and tuned 3 models within the same execution. The following two graphs display the total runtimes for these training and tuning processes across the different data sizes and frameworks (Fig. 12, Fig. 13):

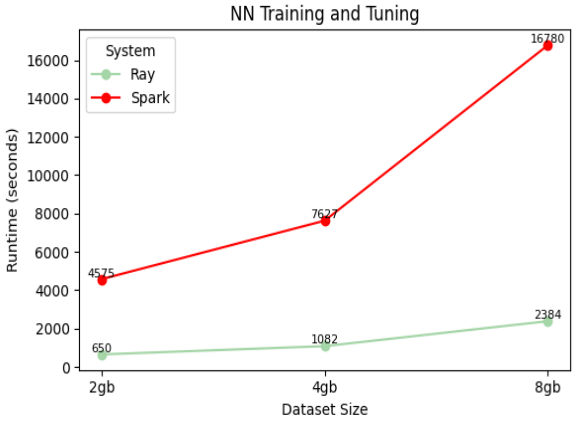

Fig. 12. NN Training and Tuning

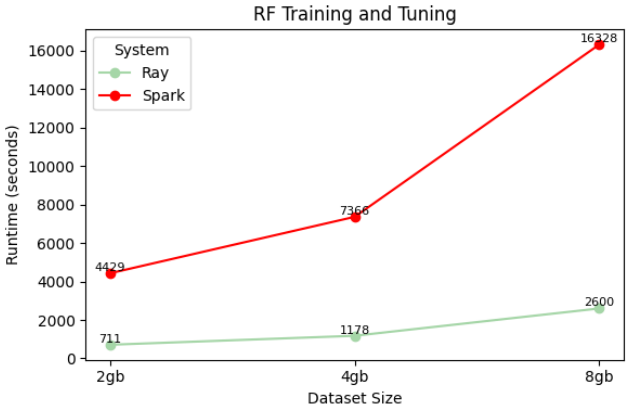

Fig. 13. RF Training and Tuning

Both models achieved solid predictive performance, with the Neural Network reaching an accuracy of 0.84 and an AUC of 0.81, while the Random Forest achieved an accuracy of 0.82 and an AUC of 0.85. The most significant performance gap between Ray and Spark lies in the hyperparameter tuning phase. As our experiments demonstrate, Ray achieves substantially lower runtimes than Spark regardless of whether the model is a Neural Network or a Random Forest—indicating that the model type has less impact than the tuning framework itself. This advantage stems from Ray's architecture, which supports fine-grained task parallelism, lightweight Python-native execution, and efficient resource utilization across nodes. In contrast, Spark's tuning process involves more heavyweight, JVM-based operations and less flexible scheduling. These differences lead to a noticeably reduced total runtime in Ray, confirming its superiority in scalable machine learning model training and tuning.

ASHRAE Competition prediction: To benchmark the performance of Ray and Spark on the ASHRAE energy prediction task, we evaluated not only execution time and CPU usage, but also the quality of the machine learning models using RMSE (Root Mean Squared Error) and R^2 (R-squared). RMSE measures the average prediction error, with lower values indicating better accuracy, while R^2 assesses how well the model explains the variance in the target variable, with values closer to 1 denoting stronger predictive performance.

A key distinction between the two frameworks lies in how they handle scale and complexity during model training. Ray leverages a modular setup using ray.data for preprocessing and XGBoost for model training, along with ray.tune for hyperparameter tuning. Due to time restrictions, this script wasn't optimized, thus any results and metrics that came from it are subjects to improvement and weren't included in this report to keep it as robust and as accurate as possible. For that reason, only the performance of Spark was taken into account for this task.

Apache Spark's MLlib pipeline with GBTRegressor, combined with CrossValidator and a parameter grid search, trains directly on the entire dataset. It benefits from Spark's mature optimization features, such as efficient memory usage, executor parallelism, and tuned shuffle operations. As a result, Spark completes full training in just a few minutes while consistently achieving low RMSE and high R^2 scores. This demonstrates Spark's superiority not only in execution speed but also in model accuracy and scalability under realistic, large-scale conditions.

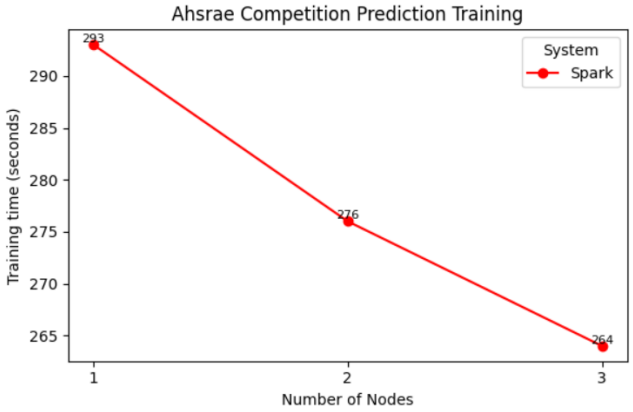

Fig. 14. ASHRAE Competition Prediction Times

VII. COMPARISON

A. *Developer Experience and Usability*

Ray offers a simpler setup compared to Spark, requiring only package installation and a few commands to connect VMs to the cluster. Both frameworks support straightforward distributed execution, but they cater to different use cases. Spark provides high-level abstractions ideal for batch processing and large-scale data analytics, albeit with a more rigid execution model. Ray, in contrast, offers fine-grained control and dynamic task scheduling, making it more flexible for machine learning and parallel computing.

Ray simplifies parallelization to distribute tasks across nodes with minimal changes to code. Spark, while more structured, provides user-friendly APIs and extensive built-in optimizations. Though Ray enables quick distribution, it lacks some of Spark's optimized operations and specialized libraries, which can be critical in certain use cases—like PageRank, where Spark's library-based implementation greatly simplifies development.

For machine learning, both platforms are developer-friendly, supporting well-known APIs such as Scikit-learn (Ray) and MLlib (Spark). In terms of dashboard support, both frameworks offer useful tools, though Ray's dashboard feels more modern and polished compared to Spark's YARN-based interface.

B. *Performance and Resource Management*

Efficient memory management is critical when working with large datasets. It ensures stability, lowers infrastructure costs, and improves execution speed. Ray provides real-time monitoring of memory usage across nodes through its dashboard, while Spark relies on the YARN interface for similar insights. In our tests, Ray consumed significantly more memory during ETL processes, often hitting performance bottlenecks. Spark, however, managed memory more efficiently, particularly during machine learning tasks. Ray frequently ran into out-of-memory issues and resorted to disk spilling, which degraded performance in those scenarios. In graph-related operations like Triangle Counting and PageRank, both frameworks performed comparably. Yet in ETL tasks, Spark dominated with significantly faster runtimes—sometimes up to 10x faster—reinforcing its suitability for data-heavy operations.

C. *Scalability and Adaptability*

Apache Spark is a mature and widely adopted framework that scales efficiently for large-scale data processing. Its architecture—based on resilient distributed datasets (RDDs)—supports robust fault tolerance and seamless horizontal scaling. Spark integrates tightly with resource managers like YARN and Kubernetes and provides structured APIs and built-in optimizations that make it easy to scale across clusters. One of Spark's key advantages is its rich ecosystem of specialized libraries, such as GraphX for graph processing and MLlib for machine learning, which dramatically reduce development time and simplify the implementation of complex distributed tasks.

Ray, by contrast, offers a more flexible and dynamic execution model that excels in parallel and asynchronous task execution. Its actor-based design allows for fine-grained control over distributed workloads, making it well-suited for machine learning, hyperparameter tuning, and other compute-intensive applications. However, Ray lacks the extensive ecosystem of prebuilt libraries found in Spark. In particular, it does not offer mature support for distributed graph processing out of the box, requiring users to implement such functionality manually. This limits its adaptability in tasks where Spark's built-in solutions, like GraphX, can handle complexity and scalability with minimal effort.

VIII. CONCLUSION

After conducting extensive experimentation with both Ray and Apache Spark, we identified clear strengths in each framework based on task type and system behavior. Apache Spark consistently demonstrated superior performance in large-scale data analytics tasks, especially when applying uniform operations across massive datasets. Its rich set of high-level APIs, mature ecosystem, and optimized execution engine made it well-suited for these workloads. Ray, in contrast, excelled in distributed machine learning, offering seamless.

Our findings suggest that Ray and Spark are not strictly competing systems, but rather complementary tools that can be used together in hybrid pipelines. For instance, Spark can efficiently process and extract features from large-scale data (e.g., computing PageRank scores or triangle counts in a graph), which can then be passed to Ray for distributed ML model training (e.g., for tasks like link prediction). While Ray's Task and Actor APIs make it easy to parallelize Python code with minimal refactoring, it sometimes lacks the mature, optimized libraries Spark offers for complex data operations. That said, Ray provides flexibility and simplicity in distributing Python workloads, especially when standard Spark abstractions are not applicable. integration with powerful ML libraries such as PyTorch and XGBoost. This makes Ray particularly effective in training and tuning complex models across distributed resources.

However, Ray comes with higher memory overhead, often leading to disk spilling or out-of-memory errors, especially in data-heavy analytics tasks. This was partly due to Ray's Object Store using only a portion of the system's RAM by default. As a result, resource tuning and careful monitoring became necessary in many cases. In conclusion, our comparative study shows that Apache Spark and Ray shine in different areas of big data and AI workloads. Spark offers unmatched performance for structured data analytics and transformations, while Ray provides a lightweight and flexible framework for distributed machine learning. Understanding when and how to leverage each tool can significantly improve the scalability, efficiency, and success of data-intensive projects.

REFERENCES

[1] Okeanos-Knossos, Greek Research and Technology Network (GRNET), okeanos-knossos.grnet.gr. https://okeanos-knossos.grnet.gr/home/

[2] D. Borthakur, "HDFS Architecture Guide," apache.org. https://hadoop. apache.org/docs/r1.2.1/hdfs design.html

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

[4] "Overview — Ray 2.41.0," docs.ray.io. https://docs.ray.io/en/latest/ray-overview/

[5] Apache Spark, spark.apache.org. https://spark.apache.org/docs/latest/

[6]NFS Server, ubuntu.com/server, https://documentation.ubuntu.com/server/how-to/networking/install-nfs/index.html

[7] snap.stanford.edu. https://snap.stanford.edu/data/