Comparison between Python scaling frameworks for big data analysis and ML: Apache Spark vs Ray

Apostolos Chatzianagnostou
NTUA ECE School
Athens, Greece
el19021@mail.ntua.gr

Panagiotis Tsikriteas
NTUA ECE School
Athens, Greece
el19868@mail.ntua.gr

Andreas Magkoutas
NTUA ECE School
Athens, Greece
el19015@mail.ntua.gr

***Abstract*—The necessity for utilizing substantial volumes of data in recent years, specifically for machine learning and neural network training, has led to the development of various frameworks. These frameworks aim to enhance the speed of different algorithms by leveraging distributed systems extensively. One widely-used framework is Apache Spark, recognized for its excellent performance. On the other hand, Ray, being a relatively new entrant, has garnered increasing attention, especially in applications necessitating machine learning capabilities. In this analysis, we seek to compare these two frameworks by evaluating their performance in different machine learning tasks. This evaluation includes not only their time efficiency but also aspects related to memory management and scalability when dealing with datasets of significant size.**

I. Introduction & Goals

One of the major challenges in the field of Data Science revolves around handling the scalability of large datasets in machine learning applications and the respective algorithms. Such algorithms require significant hardware resources, notably in terms of processing units and memory. Consequently, the noticeable trend in recent years towards establishing clusters with a substantial number of nodes is justified, aiming to meet the complexity and demands posed by big data applications. In this context, the coexistence of established frameworks like Spark, coupled with the introduction of newcomers such as Ray, naturally instigates a comparative analysis to identify their respective strengths and weaknesses. This is the primary objective of our work.

In order to address the aforementioned comparison between Ray and Spark, we initially established and configured our own cluster. This cluster consisted of 4 nodes and was implemented within our private (home) network. Further details regarding the setup and configuration will be provided in the following sections. To maintain consistency with our task, we subsequently utilized publicly available datasets of considerable size (several gigabytes) with the only limitation being the resources of our cluster machines. Whenever necessary, we also implemented custom datasets to conduct more comprehensive testing of the two systems.

This paper aims to present a comparison of the performance of both systems, primarily in executing classical algorithms and machine learning operators on big data. The evaluation mainly focuses on response time, although in some instances, the memory usage of both systems is also considered. Emphasis is placed on the scalability of the systems, evaluating their behavior under different workload sizes and varying resources. To achieve this, a lot of experiments with diverse parameters were conducted, each executed with the same configuration on both Spark and Ray. The results derived from this study are of significant interest and can decisively contribute to the selection between the two systems based on the specific task, paving the way for their more efficient utilization. Simultaneously, the results highlight inherent weaknesses and strengths of each system, particularly concerning the handling of big data. As previously mentioned, our main objective is to assess the scalability of Spark and Ray in the context of big data, rather than delving into the effectiveness of the machine learning algorithms executed on them. Therefore, we refrain from extensive commentary on the performance of the algorithms used (e.g model accuracy). The source code used for the experiments can be found at this URL: https://github.com/maGutas/BigDataProject

II. System & Software description

In this section, we aim to provide an overview of the main software components employed in the project, including Apache Spark, Apache HDFS and Ray.

A. Hadoop Distributed File System

Hadoop Distributed File System (or HDFS), is the primary distributed storage component of the Apache Hadoop framework. One significant advantage offered by this system is the easy and fast data management, coupled with fault tolerance, which is crucial for large enterprises employing it. Additionally, the system leverages the property of data locality, reducing the cost of data transfer between nodes [1].

B. Apache Spark

Apache Spark is an open-source multi-language framework suitable for scalable machine learning and data analysis code [2]. PySpark is the Python API for Apache Spark. It enables you to perform real-time, large-scale data processing in a distributed environment using Python. It also provides a PySpark shell for interactively analyzing your data. The PySpark API was utilized for implementing basic operations, while for studying machine learning operations, the highly significant Spark MLlib library was employed. MLlib is a

scalable machine learning library that provides a uniform set of high-level APIs that help users create and tune practical machine learning pipelines [3].

C. Ray

Ray is an open-source framework suitable for scaling data analysis and machine learning code. In the study of this particular domain, the primary focus involves the utilization of the API provided by Ray, known as Ray Datasets. With the assistance of this API, desired operations are executed in a distributed manner across the cluster of nodes. This proves particularly beneficial given that machine learning jobs are both time-consuming and computationally intensive [4].

III. Infrastructure Setup

We start by providing a basic set of instructions for the installation and setup of the essential systems.

A. Cluster configuration

The working environment is fully distributed and consists of a 4-node cluster, created using our personal machines (or VMs on them) running Ubuntu 22.04 LTS or Linux Mint 21 operating systems. All nodes are connected in our local wireless (home) network and they have been assigned static IPs. The total available resources of the system are :

- 20 CPUs
- 29GB RAM
- 300 Available disk space

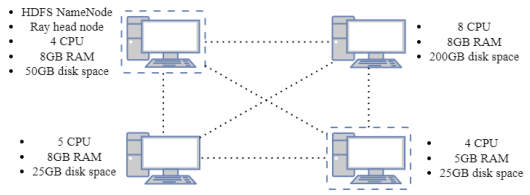

Fig. 1. Cluster topology. Dashed lines indicate Virtual Machine.

Note that the resources are not evenly distributed among the respective nodes (non-homogeneous cluster). Specifically, one node has 8 CPUs and 8GB RAM, two nodes have 4 CPUs with 8GB RAM each, while the last one has 4 CPUs with 5GB RAM. However, not all the RAM of each machine is available for the workloads, as some is reserved for running other processes, such as the operating system.

For the storage of the large volume of data, Apache HDFS, a distributed file system designed for storing large-scale datasets, was utilized. Both Spark and Ray fetch their data from HDFS. The installation and configuration instructions for the system are provided below.

B. Tools & Prerequisites

The essential tools required to appropriately set up the environment are:

- Hadoop 3.3.6 (HDFS)
- Spark 3.5.0 (Hadoop implementation over YARN)
- Python 3.10.12
- Ray 2.9.0

A main prerequisite for the correct installation of the systems is the establishment of a password-less communication among all machines through SSH. To achieve this, a process of creating an RSA public key must be undertaken.

C. Installation & Configuration of Apache HDFS

The distributed file system has been appropriately configured to utilize all available nodes, consisting of 1 NameNode and 4 DataNodes. The following steps were followed for the proper installation and configuration of the system:

- Download and extract the Hadoop 3.3.6 executable to a properly configured directory (common to all nodes).
- Export the necessary environment variables.
- Edit the configuration files to adapt the system to the available resources of the cluster. The default replication factor is set to 1.

Regarding the replication factor, which is the number of copies maintained by HDFS for each block, we choose the default value to be 1. This decision is made because the datasets to be loaded have a large size compared to the total available space. However, it's important to note a specificity of HDFS: if we attempt to upload a file (e.g., using `hdfs dfs -put`) through a DataNode and the replication factor is configured to be 1, Hadoop will try to place as many blocks as possible (ideally, all) on that specific node to maximize locality. However, this would result in non-distributed files in the cluster. To address this, before testing each script, we ensure to distribute the blocks across the cluster by temporarily setting the replication factor to 2 (easily done with the command `hdfs dfs -setrep -w 2 /path/to/file`). This way, we can be certain that copies of the blocks will be distributed to all nodes. After the tests, we revert the replication factor to 1.

Upon the successful execution of these steps, we are able to monitor and manage the file system through the graphical interface provided by Hadoop.

D. Installation & Configuration of Apache Spark

After the successful installation of HDFS, we can proceed with the installation and configuration of Spark on top of the Hadoop resource manager, YARN. The steps are as follows:

- Download and extract the Spark 3.5.0 binary for Hadoop 3.5 to a properly configured directory (common to all nodes).
- Edit the configuration files of YARN to enable the monitoring of Spark applications through graphical interfaces.
- Edit the Spark configuration file to optimize the utilization of the system resources. Specifically, we allocated a maximum of 1 GB memory for the driver and 2 GB for each executor, along with 2 CPU cores per executor. These values were determined through extensive testing.

Before executing a Spark application, activate HDFS and YARN by running `start-dfs.sh` and `start-yarn.sh` respectively. Subsequently, submit the application using the command `spark-submit <spark-app.py>`.

During the submission, various parameters can be specified, such as the number of executors and the virtual environment.

E. Installation of Ray

Ray can be installed on each node by simply runnning:

```
pip install -U 'ray[default]'
```

The first step towards setting a local Ray Cluster is to choose one of the node to act as the Head Node. On this designated node, run :

```
ray start --head --port 6397
```

Of course we are free to choose any other available port. Any other node can be added in the cluster with the following command:

```
ray start --address='HEAD_NODE_IP:6397'
```

Make sure to replace HEAD_NODE_IP with the actual head node's IPv4 address.

IV. Methodology

A. Scripts

To investigate the system, we developed three core scripts covering:

- Spark and Ray as ETL tools
- RandomForest Regression with XGBoost
- Batch inference on a pretrained NLP model

Further insights into the functionality and implementation of these specific scripts will be detailed in the subsequent sections. Each script was executed with 2, 3, and 4 workers, corresponding to an incremental augmentation of the dataset size. All the time metrics presented in the ensuing analysis represent averages derived from multiple measurements, with any outliers excluded.

B. Dataset generation

For the dataset generation, we followed the instructions provided in the assignment and searched for a publicly available dataset that meets our requirements on kaggle page [5]. The primary requirement was the dataset to be sufficiently large, ideally more than 20 GB. In this context, we eventually settled on a 90GB dataset about books in JSON format [6]. Upon closer examination, we realized that many of the included information was redundant for our use cases. Therefore, we removed anything unnecessary, resulting in a dataset of around 15 GB.

With this dataset, we successfully executed several queries related to batch processing to assess the performance of two systems in communicating with HDFS and managing large volumes of data. The generated results are provided in the following section.

Unfortunately, when we attempted to use the above dataset with machine learning algorithms such as Regression, we encountered several issues related to the resources of our cluster machines. Specifically, the dataset consisted of complex data types, such as objects, which required intensive preprocessing, that exceed the capabilities of our machine memory. We primarily faced this issue in Ray due to the inefficiency of Ray Datasets built-in functions. Additionally, this dataset lacked "useful" features for generating realistic Regression problem, significantly limiting the dataset portion size for algorithm execution to about 1-2 GB. Considering the above, we resorted to the solution described in the next paragraph.

To create user-friendly data for the ML problems we addressed, we utilized the `scikit_learn` library, specifically its `make_regression` function [7]. Based on these, we created sufficiently large datasets for Regression problems. It's worth noting that these datasets consisted solely of float values. With these specific datasets, we successfully executed our experiments and extracted useful information and conclusions.

V. Spark and Ray as ETL tools

A. Overview

Our analysis begins with the examination of the performance of Ray and Spark in ETL operations over a large volume of data. For this purpose, we utilized the previously mentioned books dataset stored in HDFS in Parquet format with a replication factor of 2. It's noteworthy that the files are not evenly distributed across the nodes of the cluster. The two machines with the highest available storage space possess the majority of the blocks. Our scripts consist of three simple queries:

- Query 1: What is the distribution of the languages?
- Query 2: For each author, find the average text reviews count and the maximum number of (registered) pages.
- Query 3: Find the 20 English books with the most pages, that include the word "book" in their description.

The first two queries investigate how Spark and Ray extract a smaller subset from an extensive dataset and operate on it through aggregation and sorting processes. The final query aims to highlight the efficiency of the two frameworks when the task involves a genuinely massive dataset. While it is true that these specific scripts are not characterized by a high degree of code complexity, their analysis has revealed several critical details regarding the mechanisms that Spark and Ray use in operations like retrieving and transforming the data.

The scripts were posed over the following three configurations:

- 2 workers, 7.5 GB
- 3 workers, 11.25 GB
- 4 workers, 15 GB

Before delving into the analysis of execution times, we need to provide some background information about the main mechanisms of Spark in query execution. This information will be helpful in the evaluation of performance later. Internally, Spark represents the dataset using RDDs, essentially a collection of distributed objects with high fault tolerance and significant capabilities for parallel processing. Any query can be implemented using RDDs by applying basic python transformation functions (map, flatmap, reduce), frequently proves to be inefficient and challenging for the programmer. For this

reason, Spark provides a higher-level API, the DataFrame API, which relies on RDDs but introduces a new abstract data type, the DataFrame. There are two main advantages to dataframes: code simplicity, since they offer an SQL-like environment and the usage of the Catalyst optimizer, a powerful tool for query optimization.

We also want to emphasize a fundamental distinction between the two scripts. For Spark, the dataset read (and some simple datatype/column adjustments) needs to occur once, at the beginning of the script. In contrast, when attempting something similar in Ray, using a single `read_parquet` for dataset reading, it led to disappointing execution times. For instance, the initial executions of queries 1 and 2 had an average execution time of about 40 minutes each (configuration: 2 workers/5GB data), even though this specific query operates on a smaller volume of data (after filtering). The problem was that Ray was reading a much larger portion of the dataset before proceeding with the necessary selection/filtering. The fix was simple: the filtering needed to be specified within the `read_parquet` function responsible for reading from HDFS! After the necessary code rearrangements, the execution time of the first query dropped to approximately 25 seconds. However, this presupposes that before executing each query, the dataset must be reloaded from the file system. The execution times for each query are discussed below.

B. Query 1

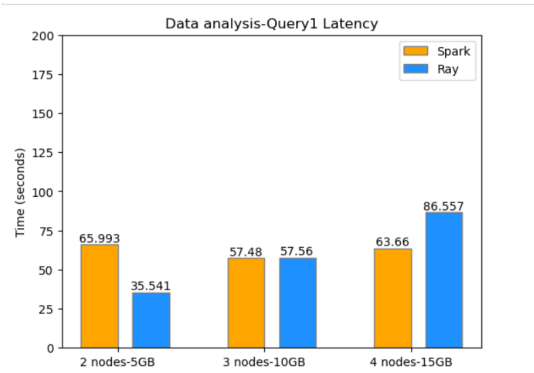

Fig. 2. Total response time of query 1

The configuration with 2 workers and a 5GB dataset is the only one, in which Spark seems to behave worse than Ray. This happens probably because in the 2 node cluster, there is not much need for block shuffling across nodes. In the other two configurations, Spark is proved to be faster than Ray, but the difference is almost negligible.

Regarding scale-up (which is shown on the above figure), Spark appears to approach the ideal (linear scale-up). The ideal scale-up is a constant line, meaning that an equal increase in workers and datasize doesn't change the total response time. In real life scenarios, the scale-up is usually a decreasing function due to factors like data transfer between nodes, etc. In our case, we observe an opposite behaviour in Spark, when we increase the number of workers from 2 to 3, the total time

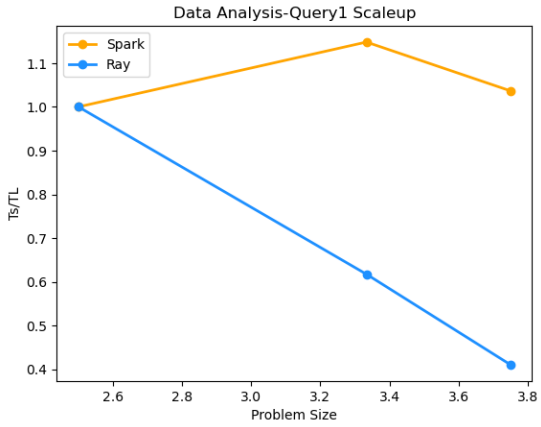

Fig. 3. Query 1 scale-up

decreases (the scale-up is increasing). We believe that this phenomenon is due to the fact that the nodes of the cluster are heterogeneous.

The first query proves that for very simple transformations, both Spark and Ray Datasets provide an acceptable solution. The only problem of Ray is that it experiences a notable delay in reading data, potentially because it doesn't take much advantage of data locality. In contrast, Spark runs over YARN (the resource manager of HDFS), which results in more efficient executor allocation based on data locality.

C. Query 2

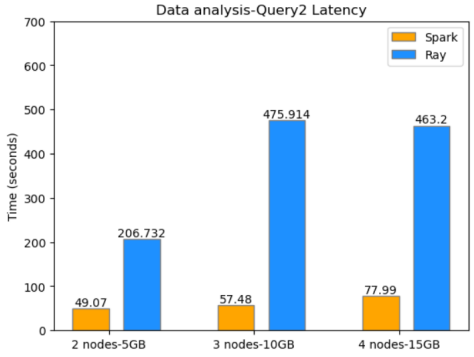

Fig. 4. Total response time for query 2

Despite not demonstrating the same consistent scale-up compared to Query 1, Spark still significantly outperforms Ray. Its performance is at least 5 times better. Here, the fact that Ray needs to read the dataset more than once, definitely affects negatively its performance. However, if this had not occurred, the execution time of Ray would be still worse. This query also highlights the ability of Spark in aggregation/sorting processes.

D. Query 3

Query 3 forces both frameworks to handle a genuinely large volume of data (the entire dataset, 15GB) in a straightforward

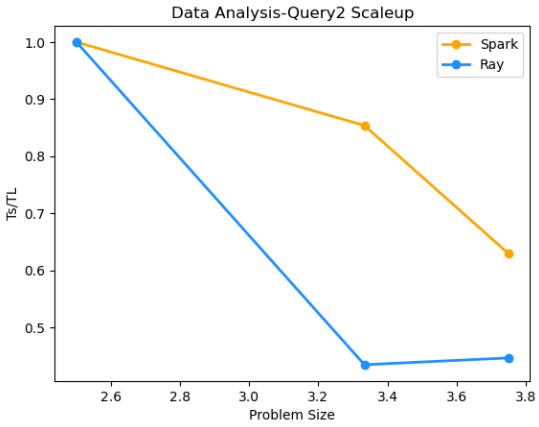

Fig. 5. Query 2 scale-up

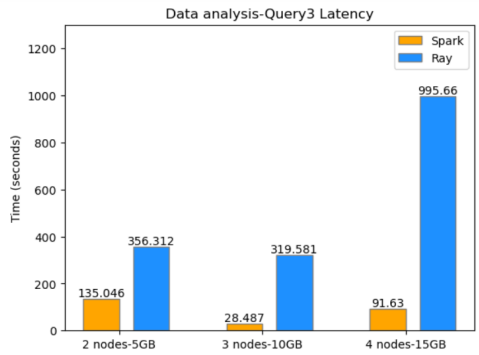

Fig. 6. Total response time for query 3

manner. Ray's scalability is significantly poor. As previously mentioned, we can potentially reduce the overall time in Ray by moving the filtering inside the `read_parquet` function. Unfortunately, such optimization is only possible for simple actions (like column selection, simple filters). In this case, as the query searches for the word "book" in each description, we are obliged to use a transformation, the `filter` function.

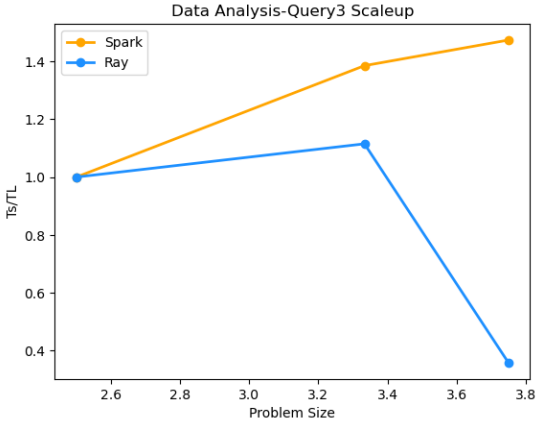

Fig. 7. Query 3 scaleup

Perhaps the main reason for the significant latency of Ray is that it lacks query optimization mechanisms and built-in, SQL-like functions as Spark does. The transformations available in Ray Datasets resemble the capabilities of RDDs in Spark, which do not perform very well on complex transformations. Another factor affecting execution time in Ray is the distribution of data across nodes. While Ray aims for maximum resource utilization, it attempts to schedule tasks on nodes that may lack the required blocks, resulting in unnecessary block transfers. The same tasks could have been executed on nodes already containing the specific blocks, fully exploiting data locality.

On the other hand, Spark exhibits excellent scalability in this particular query. The main reason for the increasing scaleup is once again the heterogeneous cluster and the distribution of blocks to all nodes.

E. Comments

Spark, one of the most recognized ETL tools, handles queries rapidly. It is worth noting that writing queries in Spark was simple and straightforward process, with no particular need for code optimization from programmer. In version 3.5, dynamic resource allocation during execution is introduced, providing flexibility without the need to manually set parameters like the total number of executors. However, this might introduce a slight overhead for executor creation and destruction.

As stated in the official documentation, Ray was not designed to become an ETL tool, explaining its reduced performance in many cases. Ray Datasets, a primary component of the Ray Ecosystem, offers basic tools for processing distributed datasets, limited to simple operations like selection, filtering, grouping, aggregation, and mapping. Unlike Spark, there is no capability for joins between two datasets. Ray also does not perform significant preprocessing for more efficient query execution, relying largely on code quality for optimal performance. However, Ray has proven unexpectedly effective in fully utilizing available resources. From the graphical dashboard we saw that the cluster's overall CPU usage often reached 100% but this does not always improves overall performance, as seen earlier.

Finally, we highlight that the parameter `parallelism` during Ray Dataset's reading appears to play a significant role in the observed latency in Ray [8]. This parameter determines how many read tasks will be created during dataset loading, indicating the number of partitions the data will be split into. However, current version of Ray, cannot create more than one task per file. Yet, if a read is followed by a transformation (e.g., map, filter), Ray will consider the parallelism value, breaking the dataset into the corresponding partitions before applying the transformation. In most cases, it is not recommended to explicitly set the `parallelism` parameter since Ray has autoscaling mechanism, but during our study, we observed that sometimes specifying parallelism reduced the overall response time. During the execution of query 3 in Ray, the default parallelism resulted in an average execution time of 995

seconds for 4 workers and 15GB of data. In contrast, setting the parallelism to 1000 reduced the time to 533 seconds. The conclusion is that to utilize Ray for data processing, code optimization by the programmer is often required.

VI. RandomForest Regression with XGBoost

A. Overview

Machine Learning Regression is a technique for investigating the relationship between independent variables or features and a dependent variable or outcome. It's used as a method for predictive modelling in machine learning, in which an algorithm is used to predict continuous outcomes [9].

Regression problems are quite common in many applications of machine learning, particularly in the field of supervised learning. The primary objective is to train a model to understand the relationships between dependent and independent variables. This stage constitutes the so-called training part of regression. Subsequently, the model is ready to take new data as input (data not present during training) and predict a specific output value, which is likely the value of a feature variable in the dataset. There are several different types of regression. In our experiments, we utilized the Random Forest Regression, making use of the respective libraries in XGBoost for both Ray and Spark.

XGBoost is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. It implements machine learning algorithms under the Gradient Boosting framework.It provides a parallel tree boosting (also known as GBDT, GBM) that solve many data science problems in a fast and accurate way. The same code runs on major distributed environment (Hadoop, SGE, MPI) and can solve problems beyond billions of examples. In addition to its numerous use cases in single-node systems, it also is also applicable across multi-node clusters. Due to its capabilities, many frameworks including Spark and Ray have integrated XGBoost as a python library [10].

Random Forest Regression is a machine learning algorithm that belongs to the ensemble learning family. Unlike traditional decision trees, which build a single tree, random forest regression construct multiple decision trees during training. Random forest regression is robust, less prone to overfitting, and can handle large datasets with ease. It's widely used in various regression tasks, including predictive modeling, forecasting, and data analysis [11].

B. Configurations

In order to study the scalability of the two frameworks, we examined the combinations of workers and dataset sizes showed on the following table. In this case, the dataset is composed from three CSV files (of 2.5, 5 and 10 GB), created using `scikit-learn`, as previously discussed. The files were uploaded in HDFS, but due to cluster's resource limitation the distribution of blocks wasn't uniform across all nodes.

TABLE I
Configurations used in XGBoost Regression

Workers	Dataset size (in GB)
2	2.5
3	5
3	7
4	10

C. Results

The following barplot demonstrates the total response time of the two systems:

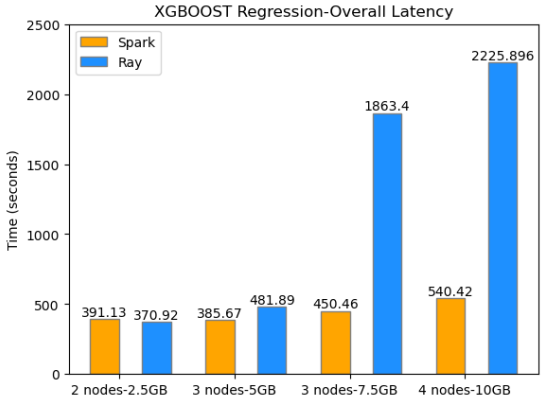

Fig. 8. Total latency (data loading, fit, predict)

We can observe that the total execution time on Ray is greater than Spark's in almost all cases. In the last two configurations (based on the arrangement of the chart), Ray resulted in significantly longer times. The main reason behind this behaviour is Ray's method of reading files from HDFS. In its current version, Ray cannot create more than one task for each file present in the dataset. This is particularly significant because, as previously mentioned, the blocks were unevenly distributed, resulting in one or two nodes in the cluster containing the majority of them. Thus, the random selection of nodes for reading, increased the likelihood of unnecessary block transfers within the cluster, resulting in longer response time. It's worth noting, however, that this issue could potentially be addressed through finer handling of Ray. For example, splitting the data into multiple smaller files could force Ray to spawn more read tasks, thus reducing the overall time. However, we couldn't find anything related to this in Ray's documentation. In contrast, Spark seemed to efficiently utilize the data locality, primarily because it runs on top of YARN, the resource manager of Hadoop.

Next, we focus on the scalability of the two frameworks specifically during training.

We notice that in all configurations, Ray consistently achieves lower execution times. Particularly, with two nodes, the difference between the two frameworks is almost threefold, which contrasts with the other cases where the difference tends to diminish. As depicted in the corresponding chart, Spark exhibits better scalability, unlike Ray, where values follow

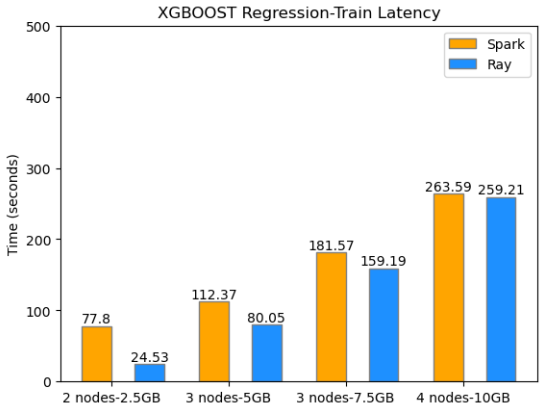

Fig. 9. Train latency (data loading, fit, predict)

Fig. 10. Train scaleup

an almost exponential curve. Notably, XGBoost `fit` function logs the training completion time, obviating the need for us to intervene in the code. Another observation arising from this particular machine learning task is that Spark often encountered memory management issues, leading to the termination of each execution. These problems were addressed to some extent by adjusting configurations, such as memory per executor (by simply changing the `spark.executor.memory` parameter before the submission). On the other hand, Ray appeared capable of managing all available memory without requiring our intervention, making the process considerably more user-friendly for developers. It is also worth noting that although both frameworks support fault tolerance via a retry mechanism, this seemed to work successfully in the case of Ray. In Spark, the loss of an executor usually couldn't be recovered even after the necessary retries.

To achieve an optimal resource utilization, we consulted the official documentation of XGBoost, which provides detailed guidance on defining critical parameters such as the number of actors/executors and CPU per actor/task [12].

VII. Batch inference on a pretrained NLP model

Before diving into the details of our final task,we believe it's essential to offer some general information about the main concepts.

A. Batch Inference

Batch inference, or offline inference, is the process of generating predictions on a batch of observations. The batch jobs are typically generated on some recurring schedule (e.g. hourly, daily). Batch inference pipelines are important because they allow for efficient and scalable inference on large volumes of data using a trained model. Batch inference pipelines are typically run on a schedule (e.g., daily or hourly) and are used to drive dashboards and operational ML systems [13].

B. Natural Language Processing

Natural Language Processing (NLP) is the sub-area of artificial intelligence (AI) that focuses on the ability to examine and understand words and texts in the same way as humans. It combines computational linguistics rule-based modeling of human language with statistical, machine learning, and deep learning models. Together, these technologies enable computers to process human language in the form of text or voice data and to 'understand' its full meaning, complete with the speaker or writer's intent and sentiment [14].

C. Transformers

Transformers are a type of deep learning neural network used mainly for NLP tasks. They are able to learn long-range dependencies between words in a sentence, which makes them very powerful for tasks such as machine translation, text summarization, and question answering [15]. In this project, we will focus on a specific NLP task, text classification, which is the assignment of a predefined category to a given text. The goal is to find the genre of a book by reading its description. Of course, we cannot build or train an entire NLP model from zero. Thus, we will use a pre-trained NLP model called BERT. HuggingFace offers an API and tools to easily download an use pretrained transformers models [16].

D. BERT

BERT stands for Bidirectional Encoder Representations from Transformers. Its is a language model based on transformers architecture. It comes in two models: BERT-base and BERT-large. For our purposes, we will use the base version, which is composed of 12 encoders with 12 bidirectional self-attention heads.

E. Overview - Task description

As we already mentioned, our task is simple: for a given description determine the category of the book. The suggested approach on this would be to get a pre-trained BERT-base model, add an extra layer and train it (fine tuning) on a subset of our dataset. But this method will not be efficient at all in our 4 node cluster, especially since no GPUs are available, since training is an extremely demanding process. Thankfully,

there is another way to achieve our goal. We can use the fill-mask pipeline from Hugging Face Transformers, that uses BERT-base model on the background. The fill-mask pipeline is simple: mask some words in a sentence and ask the model to predict them. It also allows the specification of a target list (a list of options for the masked words). So, we can add the sentence "This book is about [MASK]" at the end and set the targets to a list of book categories. Since we care about predictions, we are performing a batch inference on the fill-mask pipeline. Keep in mind that deep neural networks are extremely intensive ML models that usually require GPUs to perform faster.

F. Results

Initially, we attempted to execute the specific task on large datasets (larger than 5GB). However, in such a small and heterogeneous cluster without GPU, this process proved to be extremely time-consuming (several days), and we believe that this is not the scope of this particular analysis. Therefore, the analysis was conducted on small datasets. For reasons that will become apparent below, batch inference in Ray was performed on a set of 92,374 books, while the corresponding set in Spark did not exceed 10,000 books.

The throughput (measured in books/sec) of each framework is shown in the following plot:

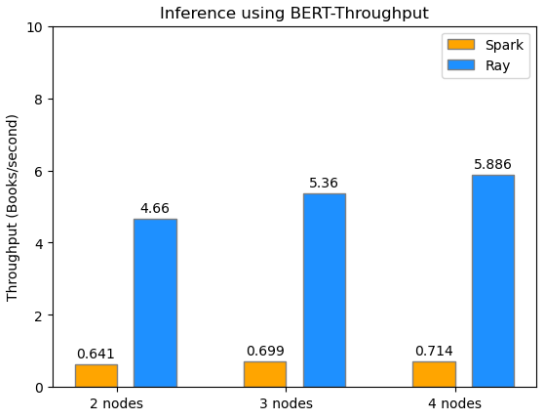

Fig. 11. Inference throughput

The goal is to maximize throughput, i.e., the number of books predicted per second. Since we used a pre-trained model without any tuning, we are not concerned about prediction accuracy.

In a CPU-intensive process like this, Ray proved to be significantly more efficient. It successfully completed predictions for a set of 92,374 books in just 5.3 hours with only 2 nodes (12 active CPUs). This translates to a throughput of 4.66 books per second. Spark failed to complete the same task in a reasonable time frame, so it was tested on a smaller dataset (10,000 books). Despite using `pandas_udf`, which handles the dataset much better than Spark's traditional UDFs, the throughput could not get past 0.73 books per second. We must admit that for this specific NLP task, the measurements

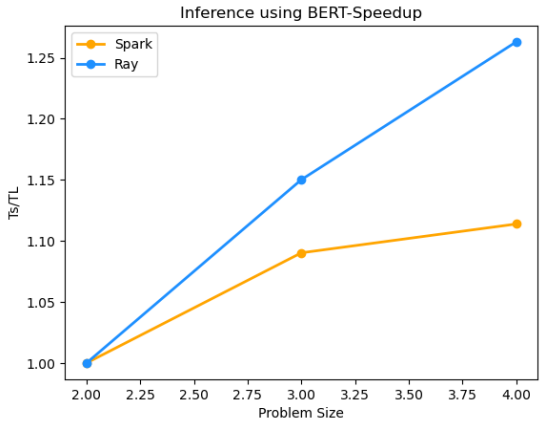

Fig. 12. Inference speedup

we obtained were not more than 3 for each combination, as it is an exceptionally time-consuming process.

By increasing the number of workers, Ray managed to achieve even higher throughput. In contrast, Spark did not show a significant difference in execution time.

The main reason for Ray's superiority is likely its significantly better utilization of available resources. In Spark, all executors have the same configuration (they must use the same number of cores, memory, etc.), leaving many system resources underutilized. Ray almost reached 100% utilization of available CPUs throughout the inference process. However, we want to emphasize that Spark's poor performance could potentially be enhanced through further code optimization, which we might not have considered. Even in such a scenario, we must acknowledge that Ray manages to scale the process much more smoothly from developer's perspective. In Ray, no special configuration was needed to fully utilize the available resources. In contrast, there were numerous occasions where we had to modify the Spark context configuration (number of executors, executor memory, shuffle partitions, etc.). This occurred because issues such as memory exhaustion, apparent underutilization of nodes, or other network-related problems related to the size of partitions (shuffle FetchFailedException) frequently arose.

VIII. Performance Influencing Factors

At this point, we find it appropriate to discuss certain factors that might have affected our measurements. One of the most notable among these was the poor quality of the local network. There were several times where the average packet delay between two nodes exceeded 150ms (as per the `ping` command output), leading to connectivity issues during execution time, especially when transferring large blocks from HDFS. Furthermore, the heterogeneity of our machines played a crucial role, particularly in terms of data storage in HDFS, where the uneven distribution of blocks was significant.

IX. Conclusion

Both Spark and Ray are undoubtedly powerful tools for managing large volumes of data and significantly facilitate the scalability of both simple and complex tasks, especially in Machine Learning applications. However, our analysis has made it clear that these two frameworks exhibit significant differences, indicating that each has been designed for a different range of applications. Spark is primarily an ETL tool, with incredible query optimization capabilities. However, this does not dismiss its usefulness in machine learning processes, as it can have a catalytic impact on the early stages of a pipeline that involve data loading and preprocessing. In this regard, Ray did not perform as well. However, Ray's main feature is its ability to maximize the use of available resources in a distributed system, even when it exhibits heterogeneous characteristics. This functionality proved to be particularly effective in the cases we examined in this work. Ray's performance, both during training and during the prediction/inference phase, exceeded our expectations. Therefore, the conclusion is that the ideal solution for optimizing the performance of a system is the combination of both frameworks. For example, in a machine learning pipeline, Spark could be used for data loading and preprocessing, while Ray could handle the distribution of training across all available resources and support the scaling up of the respective algorithm.

Acknowledgment

In the context of our work, we were provided access to three VMs to create our own cloud, which proved quite helpful for our experiments. However, for reasons beyond the scope of our study, we opted to create the cluster using our own PCs and have full control over its settings and configurations. Nevertheless, we would like to thank our instructor and the Okeanos team for their generous assistance.

References

[1] "Hadoop distributed file system (hdfs)." [Online]. Available: https://www.databricks.com/glossary/hadoop-distributed-file-system-hdfs.
[2] "Spark documentation." [Online]. Available: https://spark.apache.org/.
[3] "Pyspark overview," Sept. 2023. [Online]. Available: https://spark.apache.org/docs/latest/api/python/index.html.
[4] "Ray documentation." [Online]. Available: https://www.ray.io/.
[5] [Online]. Available: https://www.kaggle.com/.
[6] O. SKIES, "Large books metadata dataset 50 mill entries," 2022. [Online]. Available: https://www.kaggle.com/datasets/opalskies/large-books-metadata-dataset-50-mill-entries.
[7] "scikit-learn documentation." [Online]. Available: https://scikit-learn.org/stable/.
[8] "Advanced: Performance tips and tuning." [Online]. Available: https://docs.ray.io/en/latest/data/performance-tips.html.
[9] D. Castillo, "Machine learning regression explained," Oct. 2021. [Online]. Available: https://www.seldon.io/machine-learning-regression-explained.
[10] "Xgboost documentation." [Online]. Available: https://xgboost.readthedocs.io/en/stable/.
[11] "The ultimate guide to random forest regression," Sept. 2020. [Online]. Available: https://www.keboola.com/blog/random-forest-regression.
[12] [Online]. Available: https://xgboost.readthedocs.io/en/stable/tutorials/spark_estimator.html.
[13] Luigi, "Batch inference vs online inference," March 2019. [Online]. Available: https://mlinproduction.com/batch-inference-vs-online-inference/.
[14] "What is natural language processing (nlp)?." [Online]. Available: https://www.ibm.com/topics/natural-language-processing.
[15] A. Saleem, "Transformer models: The future of natural language processing," Aug. 2023. [Online]. Available: https://datasciencedojo.com/blog/transformer-models/.
[16] "Fill-mask." [Online]. Available: https://huggingface.co/tasks/fill-mask.