A Decision Support System for Automated Configuration of Cloud Native ML Pipelines

Aris Spyrou
CSLAB, NTUA
Athens, Greece
aspyrou@cslab.ece.ntua.gr

Ioannis Konstantinou
Dept. of Informatics & Telecommunications
University of Thessaly
Lamia, Greece
ikons@uth.gr

Nectarios Koziris
CSLAB, NTUA
Athens, Greece
nkoziris@cslab.ece.ntua.gr

2024 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops) | 979-8-3503-0436-7/24/$31.00 ©2024 IEEE | DOI: 10.1109/PerComWorkshops59983.2024.10502870

***Abstract*—Big Data systems like Apache Spark and Hadoop are cornerstones of large scale data processing. However, they are being utilized only on one or some steps of a larger data processing pipeline. Complex data pipelines that involve different datasets, infrastructures and programming libraries are simplified with the use of cloud-native tools like Kubernetes and KubeFlow, that model dependencies and manage the execution life cycle. Nevertheless, there are no complete solutions that are fully interoperable with Apache Spark and Kubeflow in an on-premises setup. In this work, we extend the Kubeflow Pipelines tool to support on-premises Apache Spark clusters. We experimentally evaluate our tool on an industry standard Big Data benchmark with various infrastructure and dataset configurations. We utilize the collected knowledge regarding Spark's performance to train a Decision Tree-based ML system that can detect the optimal cluster configuration according to user constraints and predict query execution time. The system can be used by non-experts through a comprehensive GUI. We finally provide open-source implementations of both Apache Spark's Kubeflow integration and the Decision Support System.**

***Index Terms*—kubeflow, cloud-native, TPC-DS, Apache Spark**

I. Introduction

Due to the large amount of data generated through various digital activities such as for example from internet applications, from electric cars, and household appliances (smart refrigerators, air conditioners, etc.), there is often a need to process this information with the ultimate aim of extracting useful knowledge. Using data analysis tools as well as machine learning techniques/algorithms one can reach useful conclusions that sometimes have commercial or research value. At the same time, a rapid growth in cloud computing has been observed in recent years, both in terms of computing and storage infrastructure. Advances in hardware technology have dropped prices and increased capabilities. This has allowed big "hyperscalers" like Amazon (AWS), Google (GCP), and Microsoft (Azure) to offer attractive and cost-effective cloud computing service platforms.

This research has been co-financed by the European Regional Development Fund of the European Union and Greek national funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the call RESEARCH – CREATE – INNOVATE (project DataSource, code:T2EDK-01231) and by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 101036388 (ZeroW)

Among other services, these platforms offer environments for creating and managing lightweight virtualization "machines" called containers. Through containers, a reduction in the project implementation times is achieved, as all modern data analytics frameworks are capable of utilizing this infrastructure (i.e., they are "containerized") and offer a rapid environment in a hassle-free manner. Google's Kubeflow [1] is such a system which runs exclusively on a container orchestration engine called Kubernetes or commonly K8s [2]. Kubeflow is a system that simplifies the implementation of ML flows in K8s frameworks, following the MLOps paradigm [3]. Among other tools, it contains Kubeflow Pipelines (KFP) [4] where developers can create data processing flows in the form of directed acyclic graphs (DAG) with each node representing a processing stage or a "component". KFP mainly targets and supports Google's GCP platform.

The Distributed data processing software ecosystem is mainly dominated by Apache Spark [5]. Apache Spark is a scalable framework capable of distributing data and computation tasks over a remote cloud or on-premises infrastructure and offers data science and machine learning packages. It supports multiple programming languages (Python, Java, Scala, R, SQL). There is now a project [6] trying to integrate Spark into the Kubernetes ecosystem to make it possible to exploit its orchestration capabilities for the Spark engine achieving faster processing and flexibility.

The main contributions of this work are the following:

- We extend KFP to offer native support for pipelines including Spark operator components while smoothly supporting on-premises or public cloud resources as well as to launch Spark workloads.
- We evaluate Spark on KFP with various analytic workloads over different computing resources using the popular TPC-DS benchmark [7].
- We train a decision tree to identify the optimal infrastructure type and size in terms of execution time or cost according to the applied workload and the user's constraints.
- We develop a decision support system (DSS) for Spark configuration on the cloud using regression techniques. We offer this tool through a graphical interface that guides the user in making decisions when it comes to choosing

resources for running workloads while respecting her constraints in terms of execution cost or time.
- Our code for both the KFP operator and the DSS is open-sourced[1].

II. Preliminaries

The aim of this section is to present a very brief background regarding the system(s) tackled in this work.

Kubernetes also known as k8s [2] is an open source container orchestration platform that automates the declaration, management and scaling of applications running in containers. The most basic term in the k8s ecosystem is the cluster. A cluster consists of machines in on-premise, public, private and hybrid cloud infrastructures. The cluster contains two parts: the control plane and the computing nodes. Each node can be a physical or a virtual machine. Each node runs "pods" and each pod contains one or more containers. The control plane runs on the master node of the cluster. The control plane consists of the kube apiserver, kube scheduler, kube controller manager and etcd modules. The compute nodes consist of the kubelet pod (the main agent that contacts the master node), the kube proxy pod (manages network-related issues) and the container runtime.

Kubeflow [1] is a suite of tools that enables data scientists to create ML workflows more easily in K8s environments. Through Kubeflow, an integrated ML solution can be created using well-known tools such as Jupyter Notebooks [8], PyTorch [9] and TensorFlow [10]. Kubeflow automates the process of installing, configuring, and maintaining the necessary dependencies so that a user can focus on building the ML model rather than the supporting K8s infrastructure. The main notion of Kubeflow is a **pipeline**. A pipeline is a description of an ML workflow including all components in the workflow and how they are combined in the form of a graph. Pipelining involves defining the job parameters required to run the pipeline and the data inputs and outputs of each component. A pipeline component is a self-contained set of code, packaged as a Docker image, that executes a step in the pipeline. For example, a component can be responsible for data preprocessing, data transformation, model training, and so on. Kubeflow pipelines is a platform for building and deploying portable, scalable container-based machine learning workflows. To facilitate the easy deployment of Kubeflow, Canonical's Juju [11] tool is utilized. Juju is a tool for managing large and complex cloud deployments. It contributes to the management of the overall deployment cycle and "connects" various deployments together in an easy way.

Apache Spark's architecture is based on the resilient distributed dataset (RDD) [12], a set of multiple read-only data items distributed across a cluster of machines, which is maintained in a fault-tolerant manner. Within Apache Spark the execution is managed as a directed acyclic graph (DAG). Nodes represent RDDs while edges represent RDD operations. Apache Spark requires a cluster scheduler and a distributed file system. The cluster scheduler can be either a built-in scheduler or another stand-alone scheduler such as Hadoop YARN [13], Apache Mesos [14] or Kubernetes. For distributed storage, Spark can interface with a wide variety of solutions, including Hadoop Distributed File System (HDFS) [15], Cassandra [16], OpenStack Swift [17], Amazon S3 [18], etc. In our setup we utilize an HDFS cluster.

[1]https://github.com/AriSpyrou/spark-kfp-hdfs-k8s

The **Spark k8s operator** is a tool developed by a team within Google aimed at integrating the Spark environment into the K8s ecosystem. The operator can be installed with the use of the helm K8s package manager tool [19] using the instructions in the relevant repository [6] maintained by Google Cloud Platform.

The Spark operator consists of:

- An event-based SparkApplication controller that listens for the creation, update, and deletion events of SparkApplication objects,
- a submit executor that runs spark-submit for submissions it receives from the controller,
- a Spark pod watcher that monitors Spark pods and sends status updates to the SparkApplication controller,
- a Mutating Admission Webhook [20] that handles customization for the Spark driver and worker pods based on the pod annotations added by the controller,

Specifically, the user uses sparkctl (or kubectl) to create a SparkApplication object. The SparkApplication controller receives the object via an observer from the API server, creates a job that takes the spark-submit arguments, and sends the job to the job executor. The job executor submits the application for execution and creates the application driver pod. At startup, the master pod spawns the worker pods. While the application is running, the Spark monitor checks the state of the application pods and sends state updates of the pods back to the controller, which then accordingly updates the application state.

III. Integrating Apache Spark with Kubeflow Pipelines

Figure 1 describes the system architecture and the interaction with a user. A user accesses Juju CLI and HDFS CLI as shown by the arrows (to the left and right of the user), while different users (e.g., administrator or developers) have access to kubectl each with specific rights. In the top we depict the k8s system modules that consist of the "computation" part of our approach whereas in the bottom we depict the "data" part of our approach. In addition, it shows the process of launching a new Spark cluster inside the stage of a KFP, then passes the request to the Spark Operator which in turn starts a driver.

A. A Cloud-native Spark Operator for kubeflow

We could follow two approaches to launch Spark on our KFP enabled k8s: the "traditional" option would be to install Spark outside the k8s environment and execute workloads by setting Spark to use the kube-api address as master. This approach would then create container workers on the cluster. The alternative that was eventually followed is to use the Spark

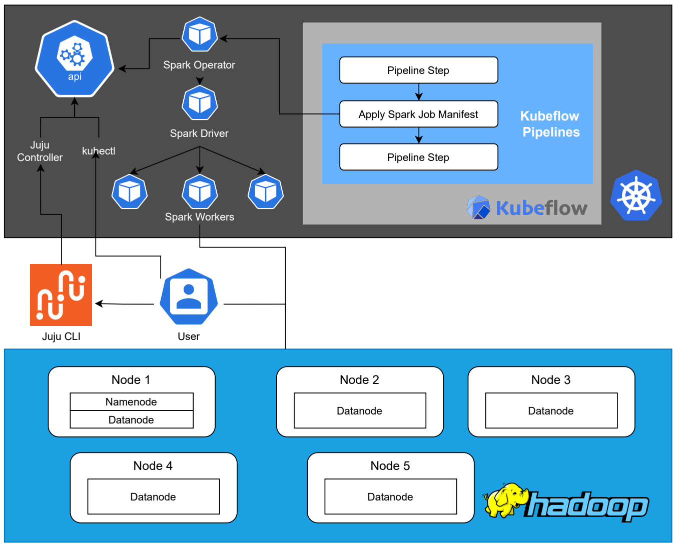


Fig. 1: System Architecture.

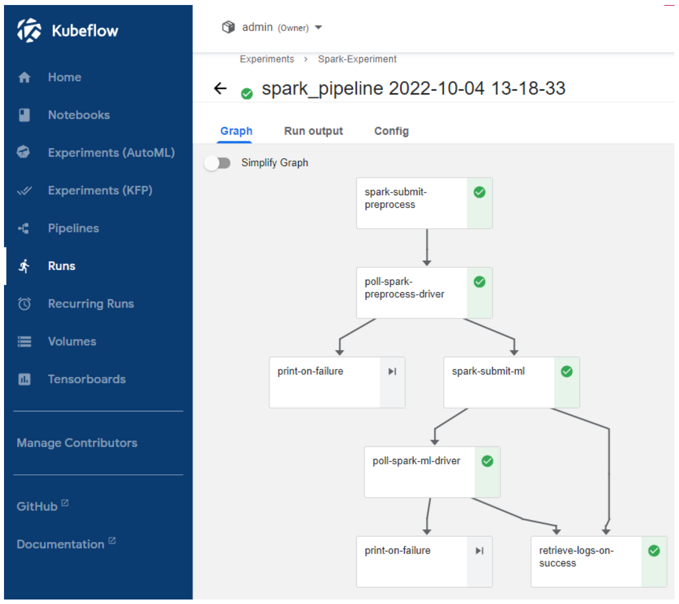


Fig. 2: An ML kubeflow Pipeline with Apache Spark

Operator described in the previous chapter that targets k8s clusters. This "cloud-native" approach is now considered a more robust way to use Spark in a cloud computing context, and especially when it comes to public clouds where the machines themselves are often not accessible.

To achieve managing a Spark workload through KFP we first need to look at how tasks are declared in the Spark Operator for Kubernetes. The declaration is made in the form of SparkApplication objects in kube-api via kubectl. Those objects are declared with the use of YAML files that contain the following job information:

- The programming language in which the main executable file was written.
- The docker image which contains a compatible version of Spark Operator for K8s and which will run in the containers that will run the code.
- The file that contains the source code to be executed. This file can be inside the image that is loaded or in some cloud storage, or in a persistent Volume [21].
- The number and type of instances, i.e workers for the workload. The important fields are "cores" and "memory" that set instance infrastructure, i.e., cores and memory.

The developed pipeline that integrates Spark on KFP consists of simple components that already exist in the basic version of KFP. The pipeline contains two main Spark task execution nodes (i.e., Master nodes), status control nodes and helper nodes. Master nodes start the Spark cluster. These are processes that read the manifest file of a SparkApplication and then use the KFP API to create new containers. At the end they return the name of the created SparkApplication as this is dynamic. Of more interest are status control nodes that manage the course of the pipeline flow. These are two nodes that were developed for the purpose of monitoring the lifetime of a SparkApplication operation. This is required since in a later stage of the pipeline another component may need to wait for the Spark workflow to complete before starting as it may depends on the Spark's results. Furthermore, because core nodes do not interact with the Spark environment, the execution state is not natively communicated to KFP. To address this, two container functions were developed that check the execution status by issuing calls to the kube-api through the kubectl tool. These functions are based on the kubectl image with version 1.21.12-debian-10-r32 of bitnami and work with the polling method. Finally, helper nodes control the result of the status check. They issue requests to the kube-api while the SparkApplication's status is "RUNNING". In practice the SparkApplication can have one of the states "RUNNING", "FAILED" or "COMPLETED". If the status becomes "FAILED" a simple error message is output while if the status becomes "COMPLETED" the helper node allows the pipeline to continue to the following nodes.

B. *A Cloud-native ML Pipeline with Apache Spark*

In order to evaluate our system we chose to run a relatively simple ML workflow. In Figure 2 we present our pipeline implementation. The pipeline uses a dataset that comes from the well-known UC Irvine Machine Learning repository. Specifically the goal is to start and manage the state of a Spark workflow within the flow context of a KFP. The data set [22] refers to data from the US population census in 1994. The goal of the ML model is the classification of individuals into two levels based on income. The training and evaluation process consists of the following stages:

- Upload to HDFS: Before any action is taken the data must be uploaded to HDFS in order to be accessible by all possible Spark worker nodes. This is the only stage that is somehow "manually" done by a client that communicates with the name node and has the files that make up the dataset on the local file system.
- Dataset Exploration: In this stage a check was made in order to become familiar with the data and to decide what

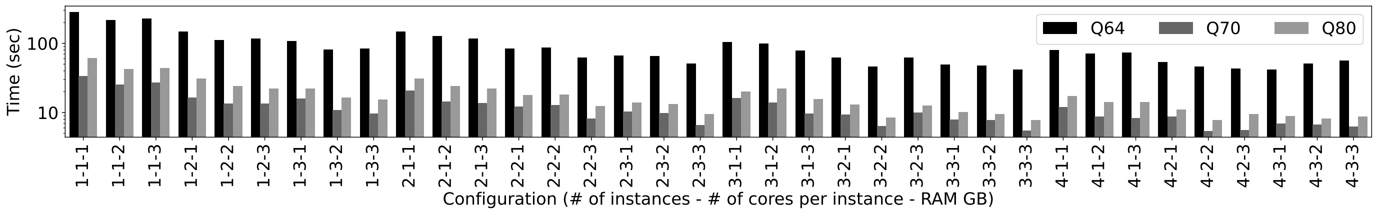


Fig. 3: Execution time of different types of TPC-DS queries for various cluster configuration sizes.

kind of transformations to apply. For the purposes of this step, an interactive Jupyter Notebook running in a Docker container on a computer outside the array was used.

- Pre-processing: Before model training, it is necessary to deal with problematic records. Processed data is cached in HDFS.
- Training: At this point a machine learning model is trained on the entire training set.
- Prediction: Finally, the model is evaluated based on its performance in the record classification from the control set.

The program is written in Python 3.9. The fully distributed framework SparkMLLib [23] was chosen to run the machine learning algorithm and preprocessing. Due to the fact that it is an application that runs inside a container, it is necessary to create an appropriate image. The main image on which the final one was based is that of the Spark Operator for Kubernetes (spark-operator/sparkpy v3.1.1) and an image was created to which the numpy library was added as it is necessary for the operation of Spark MLlib. The final image used can be found on DockerHub under the name https://hub.docker.com/repository/docker/arisspyrou/spark-py. The Python code that is required by KFP is placed in HDFS.

IV. Experimental Evaluation

In this chapter we present our experiments.

A. Evaluation Setup

As our benchmark (i.e., dataset and query workload) we selected TPC-DS [7], a well-known industry benchmark. TPC-DS is the de-facto benchmark for measuring the performance of decision support solutions, including Big Data systems. TPC-DS models retail product suppliers data. Its schema, data population, queries, data maintenance model and application rules are designed to be representative of modern decision support systems.

The initial benchmark consists of 99 different queries that examine all aspects of a system in detail to derive a more objective assessment of a system's capabilities. For this work, we chose to run 3 queries related to different types of workloads. Specifically, the queries q64, q70 and q82 were selected which are demanding in terms of network shuffle, CPU and I/O respectively, covering a wide spectrum of resource usage types.

The query execution was performed using the respective YAML manifest file and the container image file that contains the benchmark downloaded from AWS. The process is done in two parts: first, a workload creates the synthetic dataset and after this process is completed it is then possible to declare query workloads with various parameters. The synthetic dataset consists of 5 partitions with a scale factor of 10 and a parquet file type. The produced data is stored in HDFS. For the experiments, the container running the Spark driver is assigned 2 vCPUs and 3 GB of RAM. Regarding the worker nodes, 36 experiments of 3 repetitions each have been performed for greater accuracy with combinations of vCPU, RAM and instances numbers with the following restrictions:

- RAM can be 1, 2 or 3 GB as larger values can cause OOM errors.
- vCPUs can be 1, 2 or 3 as each machine has 4 and one is reserved for the smooth system operation.
- The instances can be from 1 to 4.

B. Results

We executed three queries: one that is demanding in network shuffle, one in CPU and one in I/O which will be mentioned with their codes q64, q70 and q82 respectively. q64 proved to be the most demanding overall with a range of 283 to 41 seconds with 1 worker, 1 core and 1 GB RAM and 4 workers, 3 cores (each) and 1 GB RAM configuration. On the contrary, the other two queries, q70, q82, have a range of 34 to 6 sec and 61 to 8 sec respectively.

We continue the description of our experiments following the X.Y.Z notation, where X depicts the worker number, Y the cores per worker and Z the memory in GB per worker, whereas the time is measured in seconds. As can be seen from Figure 3, as the number of workers and cores increase, the execution time clearly decreases (notice the log scale in y axis). This does not hold in the case of memory. It appears for example in q64 and in the parameterizations 4.3.1, 4.3.2, 4.3.3 (rightmost three bars) that the query time increases when we increase the memory. This cannot be attributed anywhere but to the fact that there are fluctuations in times based on external factors (e.g. background processes). Due to the fact that the data set is divided into 5 partitions and consists of a total of 10GB, each partition is around 2GB. This is confirmed in some cases such as in q64 and q82 with 1,1,X parameterization since we see that the times drop from 283 (1 GB RAM) to 215 (2 GB RAM) and 228 (3 GB RAM) and from 60.5 (1 GB RAM) to 42 (2 GB RAM) and 43 (3 GB RAM) respectively. Another interesting observation is that both the number of cores and the cluster size that the cores are distributed across are important for the query performance. For example we see that the times are 70.5 (q64), 8.6 (q70), 14 (q82) for config 4.1.2 and 85.7 (q64),

TABLE I: Pearson correlation coefficient of execution time vs different infrastructure resources

	Time q64	Time q70	Time q82
Workers	-0.636657	-0.622510	-0.632972
Cores	-0.549904	-0.540662	-0.551698
RAM	-0.132204	-0.251986	-0.251986 -0.190437

12.8 (q70) and 18.1 (q82) for parameterization 2.2.2. This improvement is about 17.6%, 32.8% and 22.1% respectively and shows that it is generally better to distribute tasks among many workers than among many worker threads.

We see that there is a general trend of exponentially decreasing execution time as the number of workers increases. We notice that as the number of workers increases, the difference between the times of the experiments with 1 core compared to the rest of the experiments in the column (i.e., with the same number of workers) decreases. Finally, we can observe that when we increase the number of cores per worker the query slows down. For example, for 1 worker we see that the difference between 1 core and 3 cores is about 176 seconds in the case of 1 GB RAM. Conversely for configurations 4.1.1 and 4.3.1 it is 38.5 seconds.

Finally, we present Table I with Pearson correlation coefficients for the pairs between workers, cores, RAM and time for each query. We notice that the number of workers is the most important factor in reducing query time. The number of cores also seems to matter. It is worth noting that we see low variation in the values in the first and second rows of the table. We notice that the RAM size does not play as big a role in reducing query time. However, what is interesting in the third row is that there is more variation between queries.

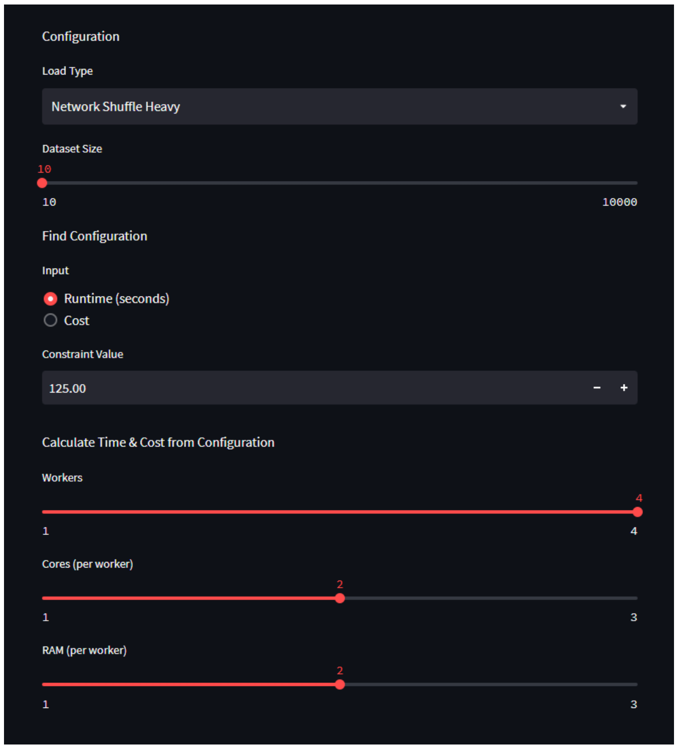
V. A Decision Support System for Workload-aware Automatic Infrastructure Configuration

After the evaluation of the system and the data collection concerning the system performance based on different combinations of settings, we develop a decision support system that utilizes this knowledge and unifies the obtained information. This system consists of:

- Database: It contains the experiments extracted information.
- Model: This is the set of input parameters that can be set and the ML model developed.
- User Interface: It allows the user to define her preferences and get the proposed configurations.

The final goal of this work is the knowledge extraction regarding the optimization of the parameterization of Spark clusters taking into account the time and the cost of an experiment.

As we have mentioned above, experiment execution times concern three different queries, each of which has different qualitative characteristics. These three represent the different input workloads. As shown in Figure 3, as the instances and cores increase, the execution time is minimized. However, the


Configuration

Load Type: Network Shuffle Heavy

Dataset Size: 10 (10 – 10000)

Find Configuration

Input: [✓ Runtime (seconds)] Cost

Constraint Value: 125.00

Calculate Time & Cost from Configuration

Workers: 4 (1 – 4)

Cores (per worker): 2 (1 – 3)

RAM (per worker): 2 (1 – 3)

Fig. 4: Decision Support System User Interface

more workers running inside Spark, the greater the overhead for the small processes that each one executes. This results in a non-linear time increase, while at the same time showing a large degree of variability between iterations as processes running in the background and other maintenance processes running by K8s affect the final flow time.

Despite the variability and wide variation observed in query executions, it was deemed appropriate to implement a system that provides support to users when selecting configurations for the workloads they run. For this purpose, a tool was developed in the form of a web app that through its graphical environment one can get a first picture of how a potential workload is going to run. In Figure 4 we present the developed User Interface.

The tool has two main operation modes. The first mode identifies the optimal configuration of Spark from the following options:

- Choice of workload type between 4 alternatives: Balanced, Network Shuffle Heavy, CPU Heavy and I/O Heavy.
- Data set size selection with 4 alternatives : 10, 100, 1000, 10000.
- Specify either a time (seconds) or cost limit (i.e., a constraint).

In this mode the system trains two decision trees on a suitable subset of the data derived from the experiments and makes an estimate for the best possible parameterization of the Spark cluster taking into account user constraints.

The second mode estimates the time and cost given a configuration in order to run a workload with the aforementioned

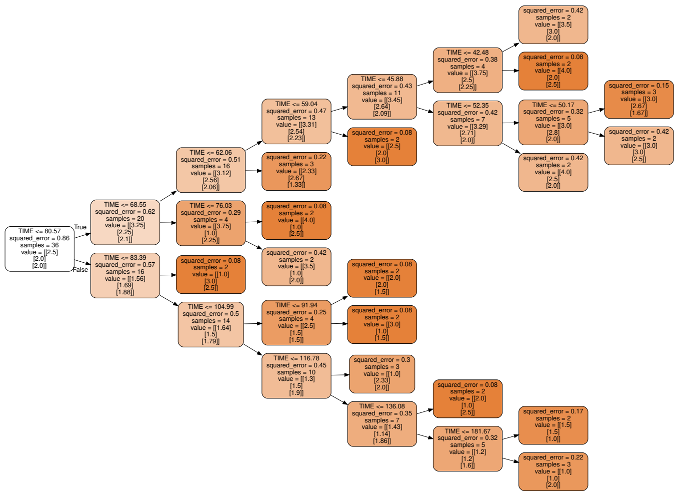

TIME <= 80.57 squared_error = 0.86 samples = 36 value = [[2.5] [2.0] [2.0]]
True
False
TIME <= 68.55 squared_error = 0.62 samples = 20 value = [[3.25] [2.25] [2.1]]
TIME <= 83.39 squared_error = 0.57 samples = 16 value = [[1.56] [1.69] [1.88]]
TIME <= 62.06 squared_error = 0.51 samples = 16 value = [[3.12] [2.56] [2.06]]
TIME <= 76.03 squared_error = 0.29 samples = 4 value = [[3.75] [1.0] [2.25]]
squared_error = 0.08 samples = 2 value = [[1.0] [3.0] [2.5]]
TIME <= 104.99 squared_error = 0.5 samples = 14 value = [[1.64] [1.5] [1.79]]
TIME <= 59.04 squared_error = 0.47 samples = 13 value = [[3.31] [2.54] [2.23]]
squared_error = 0.22 samples = 3 value = [[2.33] [2.67] [1.33]]
squared_error = 0.08 samples = 2 value = [[4.0] [1.0] [2.5]]
squared_error = 0.42 samples = 2 value = [[3.5] [1.0] [2.0]]
TIME <= 91.94 squared_error = 0.25 samples = 4 value = [[2.5] [1.5] [1.5]]
TIME <= 116.78 squared_error = 0.45 samples = 10 value = [[1.3] [1.5] [1.9]]
TIME <= 45.88 squared_error = 0.43 samples = 11 value = [[3.45] [2.64] [2.09]]
squared_error = 0.08 samples = 2 value = [[2.5] [2.0] [3.0]]
squared_error = 0.08 samples = 2 value = [[2.0] [2.0] [1.5]]
squared_error = 0.08 samples = 2 value = [[3.0] [1.0] [1.5]]
squared_error = 0.3 samples = 3 value = [[1.0] [2.33] [2.0]]
TIME <= 136.08 squared_error = 0.35 samples = 7 value = [[1.43] [1.14] [1.86]]
TIME <= 42.48 squared_error = 0.38 samples = 4 value = [[3.75] [2.5] [2.25]]
TIME <= 52.35 squared_error = 0.42 samples = 7 value = [[3.29] [2.71] [2.0]]
squared_error = 0.08 samples = 2 value = [[2.0] [1.0] [2.5]]
TIME <= 181.67 squared_error = 0.32 samples = 5 value = [[1.2] [1.2] [1.6]]
squared_error = 0.42 samples = 2 value = [[3.5] [3.0] [2.0]]
squared_error = 0.08 samples = 2 value = [[4.0] [2.0] [2.5]]
TIME <= 50.17 squared_error = 0.32 samples = 5 value = [[3.0] [2.8] [2.0]]
squared_error = 0.42 samples = 2 value = [[4.0] [2.5] [2.0]]
squared_error = 0.17 samples = 2 value = [[1.5] [1.5] [1.0]]
squared_error = 0.22 samples = 3 value = [[1.0] [1.0] [2.0]]
squared_error = 0.15 samples = 3 value = [[3.0] [2.67] [1.67]]
squared_error = 0.42 samples = 2 value = [[3.0] [3.0] [2.5]]

Fig. 5: Decision tree trained for network shuffle heavy workload on 10GB dataset. It predicts the parameterization taking as input the execution time

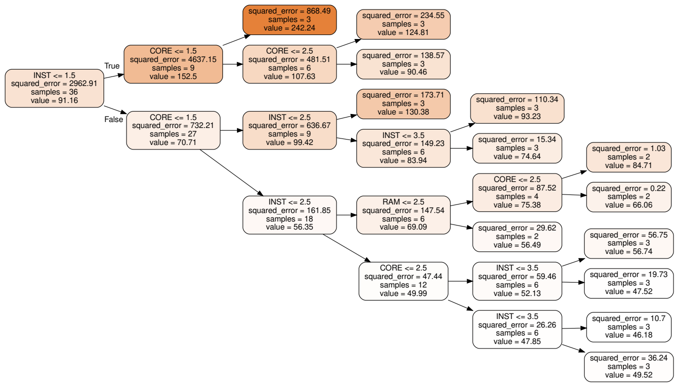

INST <= 1.5 squared_error = 2962.91 samples = 36 value = 91.16
True
False
CORE <= 1.5 squared_error = 4637.15 samples = 9 value = 152.5
CORE <= 1.5 squared_error = 732.21 samples = 27 value = 70.71
squared_error = 868.49 samples = 3 value = 242.24
CORE <= 2.5 squared_error = 481.51 samples = 6 value = 107.63
squared_error = 234.55 samples = 3 value = 124.81
squared_error = 138.57 samples = 3 value = 90.46
INST <= 2.5 squared_error = 636.67 samples = 9 value = 99.42
squared_error = 173.71 samples = 3 value = 130.38
INST <= 3.5 squared_error = 149.23 samples = 6 value = 83.94
squared_error = 110.34 samples = 3 value = 93.23
squared_error = 15.34 samples = 3 value = 74.64
INST <= 2.5 squared_error = 161.85 samples = 18 value = 56.35
RAM <= 2.5 squared_error = 147.54 samples = 6 value = 69.09
CORE <= 2.5 squared_error = 87.52 samples = 4 value = 75.38
squared_error = 1.03 samples = 2 value = 84.71
squared_error = 0.22 samples = 2 value = 66.06
squared_error = 29.62 samples = 2 value = 56.49
CORE <= 2.5 squared_error = 47.44 samples = 12 value = 49.99
INST <= 3.5 squared_error = 59.46 samples = 6 value = 52.13
squared_error = 56.75 samples = 3 value = 56.74
squared_error = 19.73 samples = 3 value = 47.52
INST <= 3.5 squared_error = 26.26 samples = 6 value = 47.85
squared_error = 10.7 samples = 3 value = 46.18
squared_error = 36.24 samples = 3 value = 49.52

Fig. 6: Decision tree trained for network shuffle heavy workload on 10GB dataset. It predicts the execution time taking as input a parameterization

options. In this mode we do not take into account the user constraints, as we want to calculate the time and cost for a given parameterization. As in the previous mode, a decision tree is trained on a subset of the data.

A. *Decision Trees*

The training process is directly related to the user's choices in the developed interface. For example, if he/she chooses a network shuffle heavy load type, then the trained decision tree will only utilize the data generated for the query q64 as its training dataset. This is done to increase the accuracy of the model as much as possible as we know in advance the type of load we want to predict. A further improvement made is to limit the leaves of the tree to achieve better generalization but also to eliminate the influence of outliers. This was done by increasing the minimum number of data per leaf to 2. Practically this means that if we want to further split a leaf then after the split the new leaves must have at least 2 values. This practice is quite common in regression. Figures 5 and 6 show two decision tree samples. Both trees are for network shuffle heavy workloads on 10GB datasets. The first tree is used to predict parameterization by setting the execution time, while the second one does exactly the opposite, i.e. it predicts the execution time based on a user-given parameterization.

VI. Conclusions

In this work we extended Apache Spark to allow its execution on a cloud-native ML pipeline through KubeFlow, a popular MLOps tool developed by Google. Using this tool, we performed an experimental evaluation of Apache Spark under various analytical workloads using the TPC-DS benchmark and different cluster configurations in a private cloud setup. The experiments provided us with valuable insights regarding Spark's behavior. We modeled those insights in the form of a Decision-Tree based DSS. We provide an open-source implementation that through a comprehensive GUI users can detect the optimal cluster configuration for given queries under constraints, or predict query execution times for various cluster configurations.

References

[1] Kubeflow. [Online]. Available: https://www.kubeflow.org
[2] Kubernetes. [Online]. Available: https://kubernetes.io/
[3] D. Kreuzberger, N. Kühl, and S. Hirschl, "Machine Learning Pperations (MLOps): Overview, Definition, and Architecture," *IEEE Access*, 2023.
[4] Kubeflow Pipelines. [Online]. Available: https://www.kubeflow.org/docs/components/pipelines/introduction/
[5] M. Zaharia *et al.*, "Apache Spark: a Unified Engine for Big Data Processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.
[6] Kubernetes Operator for Apache Spark. [Online]. Available: https://github.com/GoogleCloudPlatform/spark-on-k8s-operator
[7] R. O. Nambiar and M. Poess, "The making of tpc-ds." in *VLDB*, vol. 6, 2006, pp. 1049–1058.
[8] Jupyter Notebooks. [Online]. Available: https://jupyter.org/
[9] A. Paszke *et al.*, "Pytorch: An Imperative Style, High-performance Deep Learning Library," *Advances in neural information processing systems*, vol. 32, 2019.
[10] M. Abadi *et al.*, "TensorFlow: a System for Large-Scale Machine Learning," in *OSDI 16*, 2016, pp. 265–283.
[11] Canonical Juju. [Online]. Available: https://juju.is/
[12] M. Zaharia *et al.*, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," in *NSDI 12*, 2012, pp. 15–28.
[13] V. K. Vavilapalli *et al.*, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *SoCC*, 2013, pp. 1–16.
[14] B. Hindman *et al.*, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center," in *NSDI 11*, 2011.
[15] K. Shvachko *et al.*, "The Hadoop Distributed File System," in *MSST*. Ieee, 2010, pp. 1–10.
[16] A. Lakshman and P. Malik, "Cassandra: a Decentralized Structured Storage System," *ACM SIGOPS operating systems review*, vol. 44, no. 2, pp. 35–40, 2010.
[17] OpenStack Swift. [Online]. Available: https://wiki.openstack.org/wiki/Swift
[18] Amazon S3. [Online]. Available: https://aws.amazon.com/s3/
[19] HELM. [Online]. Available: https://helm.sh/
[20] Mutating Admission Webhook. [Online]. Available: https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers
[21] Kubernetes Volumes. [Online]. Available: https://kubernetes.io/docs/concepts/storage/volumes/
[22] B. Becker and R. Kohavi, "Adult," UCI Machine Learning Repository, 1996, DOI: https://doi.org/10.24432/C5XW20.
[23] X. Meng *et al.*, "MLlib: Machine Learning in Apache Spark," *The journal of machine learning research*, vol. 17, no. 1, pp. 1235–1241, 2016.