ORiON: Online ResOurce Negotiator for multiple Big Data Analytics frameworks

Nikos Zacheilas*, Nikolaos Chalvantzis†, Ioannis Konstantinou†, Vana Kalogeraki*, Nectarios Koziris†
*Athens University of Economics and Business, Greece
{zacheilas, vana}@aueb.gr
†National Technical University of Athens, Greece
{nchalv,ikons,nkoziris}@cslab.ece.ntua.gr

***Abstract*—In recent years we observe the rapid growth of large-scale analytics applications in a wide range of domains – from healthcare infrastructures to traffic management. The high volume of data that need to be processed has stimulated the development of special purpose frameworks which handle the data deluge by parallelizing data processing and concurrently using multiple computing nodes. These frameworks differentiate significantly in terms of the policies they follow to decompose their workloads into multiple tasks and also on the way they exploit the available computing resources. As a result, based on the framework that applications have been implemented in, we observe significant variations in their resource utilization and execution times. Therefore, determining the appropriate framework for executing a big data application is not trivial. In this work we propose *Orion*, a novel resource negotiator for cloud infrastructures that support multiple big data frameworks such as Apache Spark, Apache Flink and TensorFlow. More specifically, given an application, *Orion* determines the most appropriate framework to assign it to. Additionally, *Orion* reserves the required resources so that the application is able to meet its performance requirements. Our negotiator exploits state-of-the-art prediction techniques for estimating the application's execution time when it is assigned to a specific framework with varying configuration parameters and processing resources. Finally, our detailed experimental evaluation, using practical big data workloads on our local cluster, illustrates that our approach outperforms its competitors.**

***Index Terms*—scheduling; Big Data; resource management**

I. Introduction

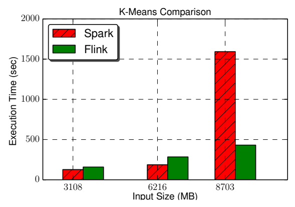
In recent years we observe the proliferation of large-scale analytics applications in various application domains ranging from traffic management [1] to financial processing [2]. The main difference from traditional analytics workloads (*e.g.,* SQL queries) is the volume of data that need to be processed. In order to be able to support the execution of such applications, novel distributed big data frameworks like Apache Spark [3], Apache Flink [4] and Tensorflow [5] have been developed. The majority of these frameworks split the processing into smaller tasks that can run in parallel on multiple computing nodes.

At the same time, cloud computing has allowed the efficient management, deployment and configuration of clusters where the aforementioned frameworks can be deployed by taking advantage of both the elastic nature of the cloud, where reserved resources are used for as long as they are needed (*i.e.*, pay-as-you-go), and the deployment/management ease of use. To exploit these two properties and fulfill enterprise needs for minimizing infrastructure maintenance and operation costs, companies follow a similar approach for either public or on-premise cloud offerings: they utilize a service that launches and manages big data clusters in order to execute the requested workloads whereas a single storage back-end is used to host data (*i.e.*, Amazon's S3, Microsoft's Azure storage, etc.).

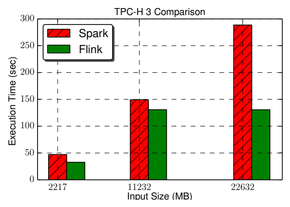
Amazon's EMR[1], Microsoft's HDInsight [6] (in collaboration with Hortonworks) and Google's Cloud Dataproc [7] are the three major public cloud platforms for managed big data deployments, whereas a typical on-premise managed service solution is Apache Ambari[2]. All these offerings support the launching and management of configurable cloud-based Big Data clusters supporting a variety of frameworks (Hadoop, Spark, Flink, Tensorflow etc.) and a variety of hardware and software configurations. Nevertheless, these frameworks offer only the necessary "plumbing" to allow configuration and management, leaving it up to the user to decide upon and configure both appropriate resources (*i.e.*, amount and type of cloud resources) and configuration settings (*i.e.*, framework and OS settings, etc.) for every workload and framework combination.

An important observation is that the current state-of-the art big data frameworks differ significantly on the way they exploit the available processing resources, the configuration parameters that can be tuned (*e.g.,* the amount of memory an application will reserve) and the policies they follow in decomposing the processing into multiple tasks. It has been documented that notable discrepencies in terms of performance can be observed between implementations of the same application in different frameworks [8]. We verify this observation in our local 8 node cluster (*i.e.*, 7 workers and 1 master node). Each node is equipped with eight CPUs (*i.e.*, Intel(R) Core(TM) i7-3770 CPU @ 3.40 GHz) and 16 GB RAM. We evaluate the performance of three widely utilized and extremely popular big data frameworks, Apache Spark, Apache Flink and TensorFlow with respect to four different applications (the applications are further described in Section V) with varying input data sizes and examine their performance in terms of execution time when executed over

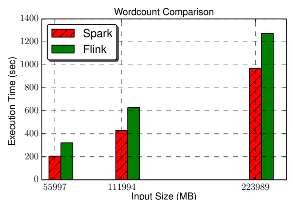
[1]https://aws.amazon.com/emr/
[2]https://ambari.apache.org/



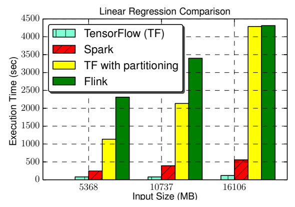
(a) K-Means.



(b) TPC-H Query 3.



(c) Wordcount.



(d) Linear Regression.

Fig. 1. Comparison of different big data frameworks for different applications.

different frameworks.

As can be observed in Figures 1(a), 1(b), 1(c) and 1(d), no framework outperforms all others for the applications under consideration. In general, Spark is more efficient than the others for most applications as it was also reported in [8] mainly due to its popularity. However, Flink and TensorFlow outperform Spark in some cases. As we illustrate in Figure 1(d), TensorFlow achieves better execution time than Spark and Flink for the linear regression application (if the input data is locally available to the worker nodes), as it is able to efficiently parallelize and orchestrate the processing. In Figure 1(b) we observe that Flink outperforms Spark for the TPC-H query. The displayed experimental results elaborate on the fact that it is beneficial for organizations to consider implementing their applications in different big data frameworks and choose the implementation which yields the best performance.

While there are multiple schedulers for orchestrating the execution of applications on specific big data frameworks [9], [10], there has been limited work in environments like the one we are describing here, where different frameworks can be deployed on dedicated clusters. Only recently has the problem of scheduling applications in such multiple cluster environments [11], [12] started to attract interest. However, the techniques presented in previous work on the subject assume that all clusters use a single framework. Additionally, they do not investigate how tuning the applications' configuration parameters can affect performance.

In this work we propose *Orion*, a novel online resource negotiator for multiple big data framework environments. *Orion*'s goal is to meet application performance requirements (expressed via deadline constraints) and at the same minimize the required monetary cost for executing them. More specifically, whenever a big data application is submitted to *Orion*, the latter determines (i) the big data framework in which the application should be executed, (ii) the time when the execution should start (*i.e.*, *Orion* may decide to stall an application until an already running application has finished and its reserved resources become available), and (iii) the tuning of the application's configuration parameters (*i.e.*, the number of reserved nodes) which satisfies its deadline constraint. More specifically, our contributions are the following:

- We propose a machine learning technique that exploits and extends regression trees by adding linear regressors at the leaf level to construct execution time cost models taking into consideration an arbitrary number of parameters which can affect the application's execution time.
- We formulate the problem of detecting the optimal configurations that satisfy submitted applications' deadline constraints as an integer linear programming problem.
- We provide a novel *online* and *framework-agnostic* resource negotiator that enables the scheduling of big data applications across multiple clusters that support different big data frameworks.
- We exploit the elasticity provided by public cloud infrastructures to dynamically change the computing resources (*i.e.*, VMs) that are allocated to the newly assigned applications with respect to the user's budget.
- Finally, we evaluate our approach using applications from different big data frameworks including Spark, Flink and TensorFlow. Our experimental results indicate the working and benefits of our approach.

II. Preliminaries

In this section we provide a brief description of the three big data frameworks we used (*i.e.*, Apache Spark [3], Apache Flink [4] and TensorFlow [5]). We note though that our resource negotiator can easily support other frameworks like Twitter's Heron [13].

Apache Spark. Apache Spark is the most widely utilized framework for big data analysis. It can support the execution of both batch and stream processing applications. The fundamental data structure that enables the in-parallel processing of the data is the Resilient Distributed Dataset (RDD) which is an immutable fault-tolerant distributed collection of objects. RDDs enable two types of operations, *Transformations* and *Actions*. Transformations lead to the creation of a new RDD while actions are operations that perform a computation on an RDD resulting to the return of a value.

There are multiple configuration parameters that can affect the performance of a Spark application. In Table I we present the parameters which mainly affect the application's execution time, the most important being the number of CPU cores reserved for the execution of an application as it controls parallelism. Furthermore, in Table I it can be observed that memory related parameters (*i.e.*, like the executors' memory) are also important since Apache Spark's main benefits arise from the efficient utilization of the available RAM. However, determining the parameters' values that minimize the application's execution time is far from trivial [14].

Apache Flink. Apache Flink is a recently developed big data analytics framework that aims at supporting the execution of both batch and stream processing applications. Flink uses a dataflow programming model that enables a record at a

Parameter	Description	Default Value
$spark.cores.max$	The number of reserved CPU cores	Not set
$spark.executor.memory$	The reserved memory	1g
$spark.shuffle.compress$	Whether shuffle data should be compressed	$True$
$spark.deploy.spreadOut$	Whether tasks should be spread out across nodes or try to consolidate them onto as few nodes as possible	$True$
$spark.shuffle.file.buffer$	Size of the in-memory buffer for each shuffle file output stream.	$32k$
$spark.speculation$	Whether to start the speculative execution of slow running tasks.	$False$

TABLE I
SPARK CONFIGURATION PARAMETERS

time processing on both finite and infinite datasets. A Flink application consists of a set of streams and transformations. A stream can be seen as a potentially infinite flow of data, and a transformation is an operation that receives one or more input streams, and generates one or more output streams. When a Flink application is assigned to a cluster, the application is mapped to a streaming dataflow with one or more sources and one or more sinks.

Moreover, Apache Flink is limited in terms of the configuration parameters that can be tuned. While the user can easily tune the parallelism of an application (*i.e.*, how many tasks will be spawned at each operation) by controlling the $-p$ parameter whenever an application is submitted, in order to change other parameters such as the reserved memory he/she must stop the cluster and update the appropriate configuration file as Flink uses the same JVM for the execution of every submitted application. Due to this limitation, we only consider parallelism degree as a tunable parameter for Flink applications. An alternative approach would be to spawn another Flink cluster for each submitted application and thus be able to set parameters like the reserved memory at the cost of having to manage multiple Flink cluster instances. We leave as future work the applicability of such an approach for tuning other parameters in Flink.

TensorFlow. TensorFlow is an open source software library for numerical computations using data flow graphs. The computation graph nodes represent mathematical operations, whereas the edges represent the multidimensional data arrays (*tensors*) communicated between them. A computation expressed using TensorFlow can be executed with little or no change in a wide variety of heterogeneous systems including commodity CPUs, or specially designed hardware accelerators, GPUs.

Designed for deep learning applications, TensorFlow implements the parameter server architecture, where a job comprises two disjoint sets of processes: stateless worker processes that perform the bulk of the computation when training a model, and stateful parameter server processes that maintain the current version of the model parameters. All instances of TensorFlow are single *python3* processes and as such, they greedily claim the resources of their hosts with no direct control on the amount of the available resources that will be used. Consequently, we only consider parallelism degree as a configurable parameter for TensorFlow applications as well.

III. SYSTEM MODEL

Our multiple clusters environment is defined via the $Clusters$ set and each cluster supports a dedicated big data framework (*e.g.*, Spark, Flink, Tensorflow). In the context of this work we make the following assumptions:

- Communication overhead for retrieving data (*i.e.*, from an external data source like CEPH[3]) is not included in the applications' execution time.
- All clusters can scale in and out (*i.e.*, dynamically increase the number of processing nodes).
- Implementations of the same application for multiple frameworks are available[4].
- Several profiling runs of the applications have been executed to gather the necessary historical data. [10].
- Finally, every application has an amount of computing resources dedicated to it from start to finish – thus, making the resources utilized by one application independent to resources used by others.

Cluster Parameters. For each cluster $c \in Clusters$ we define as fr_c the big data framework supported by cluster c. We assume that the cluster is setup in a public cloud infrastructure like Amazon's EC2 so users are charged based on the amount and type of Virtual Machines (VMs) they reserve on a per hour basis. The $cost_c$ parameter controls the amount of money that will be spent for each reserved node. Moreover, we define as $RNodes_{c,a}$ the set of nodes that have been reserved in cluster c for application a. This set is updated: i) when an application is added to cluster c and new computing resources are reserved or ii) when an application finishes and its reserved resources are released. We also store, in T_c, the points in time (*i.e.*, in ascending order) when the applications already assigned to cluster c are expected to complete their execution. Finally, the cluster administrator can optionally define a maximum per hour spending budget in the cluster via the B_c parameter.

Application Parameters. Each application a has a user-defined deadline constraint, d_a, on the application's end-to-end execution time in seconds and an st_a parameter depicting the exact system time that the application has been submitted for execution. We define as F_a the set of frameworks (*e.g.,* Flink, Spark or TensorFlow) on which the application has been implemented. For each application a we have to determine the c_a parameter which depicts the cluster that the application will be assigned. Furthermore, when an application is assigned to c_a we tune the applications' configuration parameters, tc_a, based on the big data framework it will be executed on (*i.e.*, fr_{c_a}). For instance, in Table I we present the configuration parameters that we examine when we assign an application to Apache Spark. For Flink and TensorFlow we only tune the number of reserved nodes as both frameworks require

[3]https://ceph.com/

[4]Users need to provide the source code of these implementations. Solutions such as the recently proposed Apache Beam framework [15] can be exploited to facilitate development.

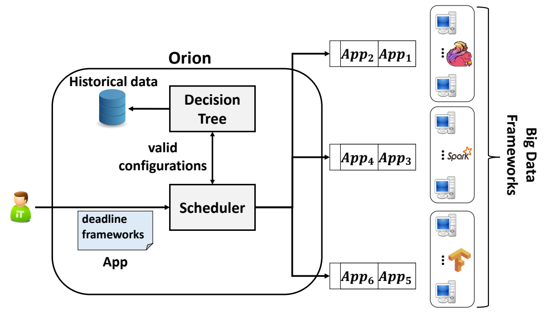


Fig. 2. Orion's High Level Overview.

restarting the cluster in order to change parameters like the amount of memory that will be utilized per computing node.

Additionally, we define, as ic_a, the set of the immutable configuration parameters for application a. We characterize them as *immutable* as they cannot be modified by the system (*i.e.*, in contrast to the parameters that comprise the tc_a set) but only by the end user. For example, this set includes the input data size and application specific parameters like the number of iterations that need to be performed in machine learning applications. Finally, the last parameter that needs to be defined is t_a, the starting time of the application in the chosen cluster. This parameter is determined based on the clusters' load and the possibility of increasing the processing resources by adding more computing nodes in the c_a cluster.

IV. Methodology

The goal of *Orion*, our novel *online* resource negotiator, is to effectively schedule big data applications across multiple big data frameworks executing on different clusters. In Figure 2 we provide the high level overview of *Orion*. Users submit their big data applications to the negotiator and the latter is responsible to determine the big data framework that will execute the application and tune appropriately the corresponding configuration parameters. *Orion* consists of two main components: (i) a *Decision Tree* based prediction approach which is used to estimate the applications' execution time over different frameworks and to suggest configuration parameters that satisfy a performance target (*e.g.*, deadline constraint), and (ii) a *Scheduler* which examines these suggestions and determines the appropriate framework and configuration according to a specific policy (*e.g.*, minimize the user's per hour spending budget). In the following sections we provide the implementation details of the two core components that comprise *Orion*.

A. Orion's Regression Tree Based Model

To be able to schedule applications across multiple frameworks successfully (*i.e.*, minimizing the number of missed deadlines) *Orion* needs a mechanism to support its decision making; one which can (i) accurately predict the execution time of a given application in various frameworks and various configuration settings and parameters and (ii) be reversely queried in order to recommend the appropriate framework and configuration settings for an application to be executed within a given time and values for the immutable configuration settings mentioned in the previous section. Using the formalization we introduced, we require our solution to be able to express the following:

$$execTime_a = f(fr_{c_a}, tc_a, ic_a) \tag{1}$$

$$(fr_{c_a}, tc_a) = f^{-1}(execTime_a, ic_a) \tag{2}$$

where $execTime_a$ is the execution time of application a given parameters fr_{c_a}, tc_a, ic_a.

In general, *supervised learning* techniques utilize a set of labeled data called training set in order to build a *"model"* of the environment which will be used to classify objects into discreet classes or predict future values based on historical ones. According to whether the class variable takes discrete or continuous values we are faced with a classification or regression problem respectively. In our case, in order to meet user requirements and utilize the necessary cluster resources without overspending, we model the environment after a regression problem where the dependent variable is the application's execution time whereas the independent ones are the application to be executed, the execution framework and all the immutable (*i.e.*, ic_a) and tunable configuration parameters (*i.e.*, tc_a). We build our model using a solution based on decision trees for regression, the details of which are presented in the following paragraphs.

Decision Trees for Regression. Classification and Regression Trees (CART) [16] constitute a very popular and easily interpretable approach for solving prediction problems using a form of supervised machine learning. Regression Trees can be employed in situations where the dependent variable is continuous, as in our case (*i.e.*, execution time). The entire environment is initially represented as the root of a single-node decision tree. We use traces of executions of the applications which are supported by *Orion* with different combinations of the parameters that are included in the model (*i.e.*, tc_a and ic_a) in order to build a training set.

During the training phase, which is repeated either periodically or when the model's accuracy drops below a threshold, the algorithm which creates the decision tree examines all measured values of all independent variables (often referred to in literature as *features* or *predictors*) in the sample's training set. The value of each tree node is calculated as the sample mean of the dependent variable of the training samples it contains. For every distinct $(feature, value)$ pair, the algorithm calculates the drop of a selected impurity metric that would be observed if the sample set were to split at that point into two distinct sets, by means of ordering on the selected value for the selected feature. In the case of a regression tree, the typical splitting policy requires the minimization of the sum of squared errors, SSE[5], as an impurity reduction metric. The algorithm then creates a splitting point using the pair with the maximum impurity decrease and recursively applies the same steps for the newly created nodes until a stopping condition

[5] $SSE = \sum_{i=1}^{n} (y_i - f(x_i))^2$, where y_i is the ith value of the variable to be predicted, x_i is the ith value of the independent variable, and $f(x_i)$ is the predicted value of y_i

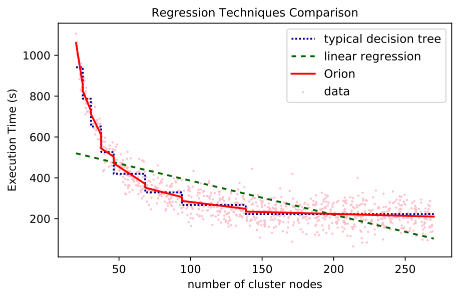


Fig. 3. A comparison between different regression techniques.

is satisfied. In contrast to more complex supervised learning methods, such as the random forest technique, regression trees produce a model which is easier to interact with by unambiguously partitioning the search space and producing a set of rules for each of the partitions, a property which is exploited by *Orion* to improve its prediction accuracy and formulate an optimization problem for each partition as explained in the rest of this section.

Linear Regression at the Leaf Level. Typical regression trees partition the search space into smaller hyper-cubes by constraining the values which different features can take as we move from the root towards the leaves. When such models are queried for predictions, the tree is traversed from the root down to a single leaf, which contains the predicted value. In essence, they build piecewise constant models by partitioning the initial state space into sub spaces. For each of these partitioned spaces, the sample mean of the dependent variable is calculated and assigned as the predicted value for that specific partition.

In order to acquire more accurate predictions, we enrich the typical regression tree technique by building linear regression models and applying them to the leaves of the tree, as introduced in [17]. This allows us to make more fine grained predictions, also taking into account information which the regression tree has not utilized, as all features are used for the creation of these models – not just the ones which participate in the tree. To visualize the impact of this technique, we present a typical non-linear curve of how execution time scales in relation to the number of nodes in a distributed system, where the initial performance gain due to parallelization drops when too many nodes are added, consistent with [18]. We test our tree based solution by comparing it with the two techniques it borrows from, simple linear regression and a piecewise constant decision tree. Adding random noise, we try to fit the curve as well as possible. The results are presented in Figure 3, where one can observe that *Orion*'s solution fits the data more smoothly than either the simple decision tree or the linear regression model. We should note that while our approach in most cases improves a naïve regression tree's predictions, it might be vulnerable to outliers in the training set which can deteriorate its performance, as we further discuss in Section V.

Search Space Partitioning. Having implemented the previously described model, we can answer queries regarding execution time predictions for specific applications in given frameworks and configuration parameters. This, however, is not enough for the enforcement of a scheduling policy. We need a way to query our model for the framework and optimal configurations under which applications can be executed while respecting execution deadline constraints. This process involves searching across all possible combinations of meaningful values for the different features in our model for solutions which satisfy user requirements. To make the solution of such a problem feasible, there is a strong requirement for limiting the search space as much as possible, even more so in cases where the number of features is large since the complexity increases in an exponential manner in relation to dimensionality.

We can recognize four distinct sources which dictate boundaries in the values of the features we are working with, thus shrinking the search space. Those are the following: (i) *Logical constraints.* All of the features we use (*i.e.*, number of nodes in the cluster, memory capacity etc.) take positive integer values, usually following specific patterns (*i.e.*, VMs usually come with 2/4/6 cores, 2/4/6/8 GB of memory etc.), (ii) *Infrastructure specific constraints.* These are closely related to the underlying infrastructure our clusters are deployed on. We make the assumption that it is not desirable for computing resources to arbitrarily scale beyond a certain point. These kinds of constraints help us apply upper boundaries to most of our features (*i.e.*, the total number of nodes cannot exceed a fixed number). (iii) *Regression tree generated constraints.* These constraints come in the form of rules, describing our decision tree. Each leaf has been produced from the initial state following a certain set of rules which differentiate it from the rest. *Orion* needs to search every sub space, represented by a leaf, for solutions to the problem in hand. This search will strictly be performed within the boundaries of each partition itself. (iv) *Human input.* For every workload there are user-defined immutable features (*i.e.*, input size, batch size, etc.) which affect both the execution time and cost.

When a user submits a job to *Orion*, the immutable features are evaluated on-the-fly. *Orion* uses this information to narrow the search space even further. It is obvious that constraints which fall into (i), (ii) and (iv) have a global effect whereas constraints which fall into (iii) affect only their respective space partition. For each of these hyper-cubes, there exists a set of constraints which significantly restrict the complexity of the search for solutions. Formulating the problem as a piecewise linear function where all parameters in Equation 1 are included into vector $\mathbf{X}$, we get:

$$execTime_a(\mathbf{X}) = \begin{cases} \alpha_0 + \sum_{i=1}^{n} \alpha_i x_i, & \mathbf{X} \in A \subset S \\ \beta_0 + \sum_{i=1}^{n} \beta_i x_i, & \mathbf{X} \in B \subset S \\ \vdots \\ \xi_0 + \sum_{i=1}^{n} \xi_i x_i, & \mathbf{X} \in \Xi \subset S \\ \vdots \end{cases}, \tag{3}$$

where $S \equiv (A \cup B \cup \ldots \cup \Xi \cup \ldots)$ is the initial state space, $\mathbf{X} = \left[fr_{c_a}, tc_a, ic_a\right]$ the vector of independent variables of dimension n and α_i, β_i, etc. are the coefficients of the linear regression models corresponding to the regression tree leaves.

The search space partitions $A, B, \dots \Xi \dots$ are defined by the aggregation of the constraints analyzed above.

Integer Linear Problem Formulation. The motivation behind this work is to propose a solution to the problem of scheduling applications across multiple clusters in a way which will allow them to respect deadlines while also minimizing a utility function, $C(\mathbf{X})$, which represents cost and is aligned with the scheduling policy. The cost function will be further discussed in Section IV-B. Going back to our analysis so far, we observe that multiple solutions might appear in either the same or different partitions of the initial space. These solutions need to be located and evaluated, in terms of the cost function. Our problem can be broken down into a number of sub-problems equal to the number of leaves in our regression tree model. As previously highlighted, the features of our models (*i.e.*, number of cluster nodes, memory capacity, buffer size) take integer values. This allows us to introduce the following *integer linear programming*[6] problem [19]:

$$\begin{aligned} \textbf{minimize} \quad & C(\mathbf{X}) \\ \textbf{subject to:} \quad & execTime_a(\mathbf{X}) < D, \\ & \mathbf{X} \in S_k \subset S, \\ & \mathbf{X} \in \mathbb{Z}^n, \end{aligned}$$

where S_k is each of the partitions of the search space $(A, B, \dots \Xi \dots)$ and D the maximum execution time we can afford. To find the desired configurations, we need to solve this problem for every partition, collect the partial solutions and evaluate them. The problem formulation allows us to utilize SCIP [20], a state-of-the-art non-commercial software for constraint programming, mixed integer programming, and satisfiability modeling and solving techniques. SCIP's approach is built around a branch-and-bound approach, along with many optimizations. Branch-and-bound is a general method for finding optimal solutions to typically discrete problems.

The algorithm [21] searches the entire space of candidate solutions by excluding large parts of the search space using previous estimates on the quantity being optimized. The worst case performance of such an approach is equal to an exhaustive search over all possible solutions, however in the cases we encountered, solutions were produced within 0.1 second. To start solving the problem presented above for all leaves, we first have to reversely traverse the tree (leaves to root) and extract the constraints which define each partition, consistently with our precious analysis. This process is presented in Algorithm 1 and only needs to happen once after the creation of the tree. Subsequently, we add the global constraints to the leaf-specific ones for the definition of the problem and solve each sub-problem separately. The solutions are then gathered, aggregated and evaluated by the *Orion*'s Scheduler component which is described in more detail in the next section.

B. Orion's Scheduler

The goal of the *Orion*'s Scheduler is, given an application a, to determine the cluster c_a that the application will be assigned to, its starting time t_a and the tc_a configuration parameters in order to satisfy the application's deadline (*i.e.*, d_a) and at the same time minimize a utility function. Different utility functions can be used for determining the appropriate assignment of an application to a cluster. For example, we can assign the application to the cluster that minimizes the application's execution time [9] or leads to the smallest increase on the user's spending budget [22]. In this work, we decided to consider the minimization of the user's spending budget as optimization goal. We envision multiple applications being submitted to each cluster so it is highly important to allocate computing resources for each application optimally.

[6]An integer linear programming (ILP) problem is a mathematical optimization or feasibility program in which some or all of the variables are restricted to be integers and the objective function and constraints are linear.

Algorithm 1 Rule set extraction

```
1: Input: $leaves$: The set of all leaves in the tree.
2: function RULEEXTRACTOR($leaves$)
3:     for each $leaf \in leaves$ do
4:         $current\_node \leftarrow leaf$
5:         while $current\_node.has\_parent()$ IS $TRUE$ do
6:             $rule\_set_{leaf}.append(current\_node.rule())$
7:             $current\_node \leftarrow current\_node.parent()$
8:     return $rule\_set_{leaf}, \forall\ leaf \in leaves$
```

Our scheduler exploits the decision tree technique described in Section IV-A to estimate the execution time of the application and suggest candidate solutions (*i.e.*, all the solutions to the optimization problem introduced in Section IV-A for each of the search space partitions). Applications might not always start executing immediately upon submission for various reasons (*i.e.*, cluster congestion, scheduling policy *etc*). Therefore, based on the estimations on the application's execution time we use the notion of slack [10] to capture whether the application will be able to meet its deadline taking into consideration its deadline and the time its execution starts. The slack metric depicts how close the estimated execution time is to the user-defined deadline. More formally, we define the $slack_a(fr_{c_a}, tc_a, ic_a, d_a, t_a)$ metric for a chosen assignment as:

$$\begin{aligned} slack_a(fr_{c_a}, tc_a, ic_a, d_a, t_a) = d_a - execTime_a(fr_{c_a}, tc_a, ic_a) \\ + st_a - t_a, t_a \geq st_a \wedge t_a \in T_{c_a} \cup \{st_a\} \end{aligned} \tag{4}$$

Matching this formalization to the optimization problem presented in the previous sub-section, we set $D = st_a + d_a - t_a$. Thus:

$$execTime_a < D \Leftrightarrow slack_a > 0 \tag{5}$$

Small positive slack values indicate the application is close to missing its deadline, while negative values indicate it is bound to miss its deadline. The slack metric depends on the time t_a, when execution begins. Ideally, if the application starts as soon as it is submitted ($t_a = st_a$), then the slack is simply computed as the difference between the deadline and the estimated execution time. However, it is possible to stall the execution of an application (*i.e.*, by setting $t_a > st_a$) until some of the nodes reserved by other applications in cluster c_a are available. In such cases, we incorporate this delay (*i.e.*, $t_a - st_a$) by adding it to the computation of the slack metric.

Furthermore, a utility function (*i.e.*, objective or cost function) which expresses how scheduling decisions affect a user's spending budget needs to be decided upon. Without loss of

Algorithm 2 Orion Scheduler

```
1: Input: $a$: the submitted application, $d_a$: the application's deadline, $Clusters$: the set of available clusters, $dt$: the decision tree that will be used for estimating the application's execution time.
2: Output: $c'$: the cluster where the application will run, $t'$ the point in time when the application will start, $tc'$: the tunable configuration parameters' values.
3: $minBudget \leftarrow \infty$; $t' \leftarrow -1$
4: for all $c \in Clusters$ do
5:     if $fr_c \notin F_a$ then
6:         continue
7:     $T \leftarrow T_c \cup \{st_a\}$
8:     for all $t \in T$ do
9:         $s \leftarrow dt.findSolution(c, a|slack > 0)$
10:        if $s.b > B_c$ then
11:            continue
12:        if $(s.b < minBudget)$ then
13:            $c' \leftarrow c$
14:            $t' \leftarrow t$
15:            $execTime \leftarrow s.execTime$
16:            $tc' \leftarrow s.tc$
17:            $n' \leftarrow s.nodes$
18:            $minBudget \leftarrow s.b$
19: if $t' == -1$ then
20:     return NULL
21: $T_{c'} \leftarrow \{t \in T_{c'} \wedge t > t'\}$
22: $T_{c'} \leftarrow T_{c'} \cup \{t' + execTime\}$
23: $RNodes_{c'} \leftarrow RNodes_{c'} \cup \{n'\}$
```

generality we consider the simple scenario where our clusters are homogeneous. The cost in that case depends on the number of nodes that are utilized by the cluster, the per hour fee charged by the service vendor and the application's execution time. More formally, we compute the cost of executing application a on cluster c_a:

$$b = cost_{c_a} \times RNodes_{c_a,a} \times execTime_a \tag{6}$$

We use the budget cost function b as the utility function C introduced for the optimization problem.

Refining the Optimization Problem. Given a set of clusters $Clusters$, where each cluster c supports a different big data framework fr_c and an execution deadline d_a our goal is to determine the cluster c_a to which a submitted application a should be assigned to, the time t_a its execution should start and the optimal configuration parameters tc_a. Using the notion of slack we have introduced, the initial optimization problem can be expressed as follows:

$$\begin{aligned}\textbf{minimize}\ & b(c_a, tc_a, ic_a, d_a, t_a)\\ \textbf{subject to:}\ & slack_a(fr_{c_a}, tc_a, ic_a, d_a, t_a) > 0\\ & b(c_a, tc_a, ic_a, d_a, t_a) < B_{c_a}\\ & fr_{c_a} \in F_a\end{aligned}$$

The goal of the optimization problem is to assign the application to the cluster that will lead to the minimum spending budget and at the same time will maximize the probability that the deadline constraint is satisfied. We propose a greedy algorithm for solving this problem (*i.e.*, Alg. 2). The algorithm receives as input the available clusters, the big data application to be executed, its deadline and the decision tree based model. It examines each cluster and checks all the valid timestamps when the application may start. These timestamps correspond to points in time when the execution of applications already assigned to the cluster have finished, thus the available resources can be utilized by the newly submitted application. We include the application's submission timestamp into this set (*i.e.*, T in Alg. 2) because we want to be able to start the application's execution immediately without waiting for the scheduled applications to finish if its deadline cannot be satisfied otherwise. The algorithm examines these timestamps in ascending order and for each timestamp utilizes our proposed decision tree based approach (see Section IV-A for more details) to estimate the appropriate configuration parameters which satisfy the deadline constraint and minimize the cost function for each partition of the initial search space. The decision tree model will return a valid combination of the configuration parameters. Then the Scheduler will consider these parameters and cluster only if they minimize the cost (see Equation 6) without violating the budget constraint. Finally, *Orion* assigns the application to the framework which yields the minimum cost and updates the timestamps set and reserved nodes for the selected cluster.

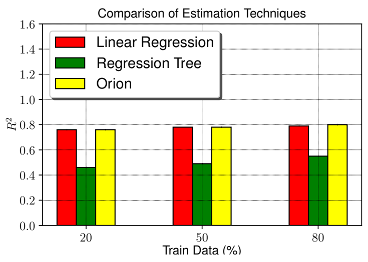
Application	Dataset	Input Data	d_a
Linear Regression	Year MSD [23]	5 GB	500
ALS	User's ratings [24]	500 MB	100
K-Means	Random Points [25]	6 GB	350
TPC-H 3	Random Data [26]	20 GB	200
TPC-H 10	Random Data [26]	20 GB	250
Wordcount	Amazon reviews [27]	106 GB	500
Grep	Amazon reviews [27]	106 GB	400
Pi	Random Samples	$100,000$	600

TABLE II
APPLICATIONS' DESCRIPTION

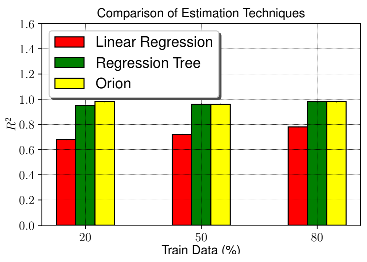
V. EVALUATION

Setup. We conduct an extensive experimental evaluation considering three different clusters. Each cluster runs a dedicated big data framework (either Spark, Flink or TensorFlow) and comprises 1 Master and 7 worker nodes. Each node is equipped with eight CPU processors (*i.e.*, Intel(R) Core(TM) i7-3770 CPU @ 3.40 GHz) and 16 GB RAM. We use Spark 2.0.2, Flink 1.4.0 and TensorFlow 1.3.0. For the evaluation of *Orion* we consider eight big data analysis applications used in similar settings [8], [28] (see also Table II). Applications are submitted in *Orion* in a round-robin fashion and the applications' submission time follows a Poisson distribution. For applications assigned to Spark, we consider as tunable parameters (*i.e.,* tc_a in Section III) the ones illustrated in Table I. In contrast, for applications assigned to either Flink or TensorFlow we use the per application reserved CPU cores as the sole tunable parameter due to the limitations we explained in Section II. We set the reserved memory of these frameworks to 12 GB per worker node. Our goal in the experimental evaluation is to measure the performance of our techniques in terms of: (i) *accuracy* in estimating the applications' execution times, (ii) the number of *deadline violations* as we increase the number of applications submitted to *Orion* and (iii) the number of *reserved nodes* during the course of the experiments as they are an indication of the user's expected spending budget.

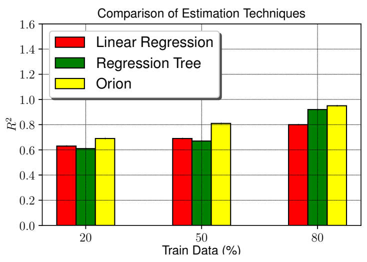
Prediction Models Comparison. In the first set of experiments we evaluate how well *Orion* captures the applications' execution times. As already explained in Section IV, *Orion* exploits the use of a decision tree based solution to estimate the



(a) Spark.

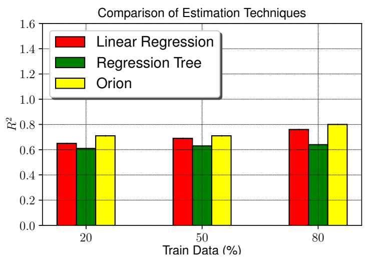


(b) Flink.

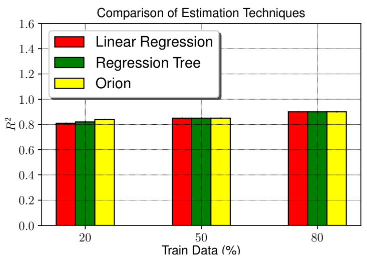


(c) TensorFlow.

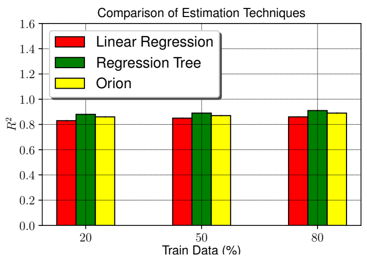
Fig. 4. Comparison of the different estimation techniques for the regression application implemented on the different big data frameworks.



(a) KMeans.



(b) TPC-H 3.



(c) TPC-H 10.

Fig. 5. Comparison of the different estimation techniques for different Spark applications.

execution time of the applications. We compare our solution against linear regression and a piecewise constant decision tree. In Figures 4(a), 4(b) and 4(c) we evaluate the accuracy (*i.e.*, R^2 score) of the estimations for the Linear Regression Application in the three different frameworks and three different prediction models when we vary the percentage of data that is used for training. We vary the training data percentage from 20% to 80% and evaluate the accuracy of the models using the remaining data points (*i.e.*, test data). The results prove that *Orion* in most cases outperforms the alternative techniques while is, at worst case, on par with them. *Orion*'s tree based solution consistently performs well compared to the poor accuracy of linear regression and the typical regression tree for some of the benchmarked frameworks (Figures 4(a), 4(b) respectively). In Figures 5(a), 5(b), 5(c) we illustrate the accuracy of the predictions for the Spark implementations for three applications (results for other applications omitted for lack of space). *Orion* still outperforms the other techniques with the exception of the TPC-H query, where regression yields better results. This can be attributed to outliers having a negative impact on *Orion*'s performance by affecting the model's linear regressors.

Comparison against other Scheduling Techniques. In the second set of experiments we compare *Orion* against a workload agnostic baseline (*i.e., Least Loaded* in the figures), which assigns the application to the least loaded framework (in terms of assigned applications) reserving for each application all the available nodes. More specifically, whenever a new application is submitted to the *Least Loaded* negotiator, the latter assigns the application to the framework with the least running applications. As we illustrate in Figures 6 and 7, *Orion* outperforms the *Least Loaded* negotiator as it is able to satisfy more deadline constraints and requires less nodes. In Figure 6 we observe that *Orion* is able to satisfy 70% deadline constraints even when 75 applications have been submitted. In contrast, *Least Loaded* manages to satisfy only 31% of the deadline constraints. The results are as expected, as Least Loaded does not consider the performance of the application on the available frameworks when it makes an assignment. Furthermore, it can be observed in Figure 7 that *Orion* achieves better resource utilization than the *Least Loaded* technique as it increases the computing nodes only in cases that it is truly beneficial. We elaborate further on the reserved resources per framework in Figure 11 where we show the per framework reserved nodes. *Orion* schedules applications to all three frameworks and increases the resources only if it is necessary for avoiding a deadline violation.

Evaluating different Cluster Setups. In order to illustrate the benefits of using a multiple frameworks environment we compare it against two alternative infrastructures. The idea is to reserve all the available nodes (*i.e.*, 21 nodes in total) initially for a Spark cluster and then a Flink cluster. The scope of the experiment is to examine whether it is helpful to have smaller sized clusters comprising different frameworks or a single cluster of one framework type (either Spark or Flink). We can see in Figures 9 and 10 that our infrastructure is able to meet significantly more deadline constraints as it exploits all frameworks, due to the fact that some applications execute better on specific frameworks (*e.g.*, Linear Regression has better results when executed over TensorFlow as we illustrate in Section I). Furthermore, Figure 11 showcases how our technique exploits the TensorFlow framework for the execution of the Linear Regression applications allowing it to use less resources than the other setups.

Impact of Applications' Inter-Arrival Times. In the last set of experiments we evaluate the deadline violations (*i.e.*, Figures 12 and 13) and the reserved resources (*i.e.*, Figure 14) when we vary the applications' inter-arrival times in seconds. As we illustrate in the Figures the lower the inter-arrival time the harder it is to satisfy the deadline constraints and also

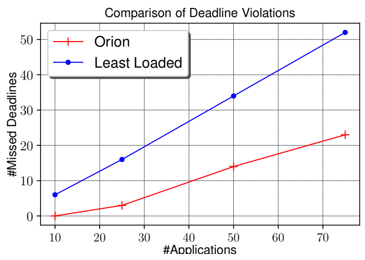


Fig. 6. Comparison of different scheduling techniques in terms of deadline violations.

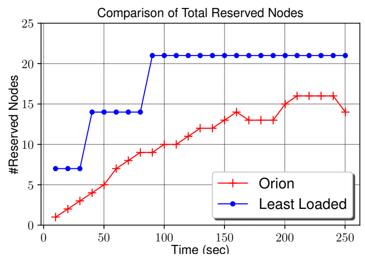


Fig. 7. Comparison of different scheduling techniques in terms of reserved resources when 50 jobs have been submitted.

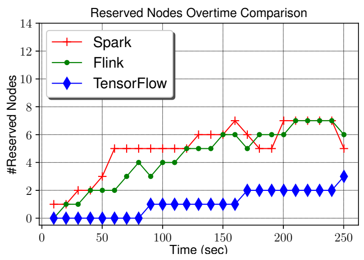


Fig. 8. The number of nodes reserved by the different frameworks when 50 jobs have been submitted.

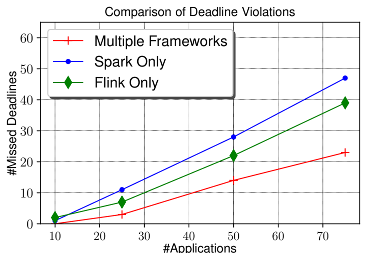


Fig. 9. Comparison of the total deadline violations for varying number of applications.

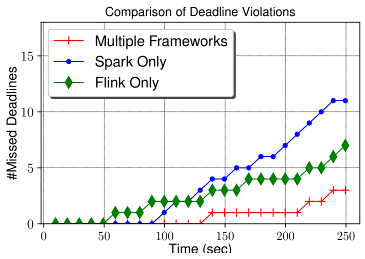


Fig. 10. Comparison of the deadline violations over time on the different clusters when 50 jobs have been submitted.

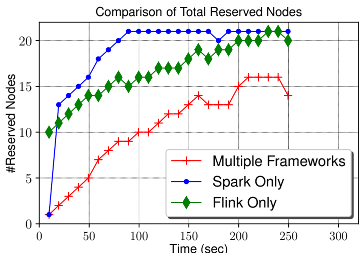


Fig. 11. Comparison of the total reserved nodes on the different clusters when 50 jobs have been submitted.

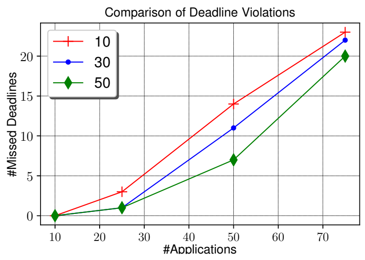


Fig. 12. Impact of inter-arrival times on the deadline violations for varying number of applications.

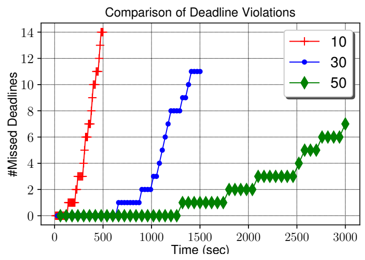


Fig. 13. Impact of inter-arrival times on the deadline violations over time when 50 jobs have been submitted.

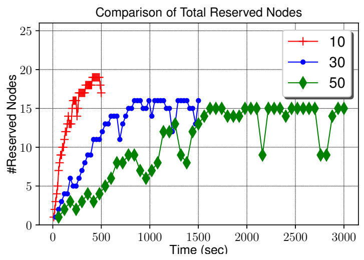


Fig. 14. Impact of inter-arrival times on the number of reserved nodes over time when 50 jobs have been submitted.

the more resources need to be reserved. This is expected, as when the application submission rate increases, it becomes harder to satisfy all the deadline constraints. Nevertheless, as we illustrate in Figure 12, *Orion* satisfies 70% of the assigned applications' deadlines even when the inter-arrival period is set to 10 seconds with more than 75 submitted applications.

VI. Related Work

There have been multiple previous works that tackle the problem of scheduling big data applications in distributed systems with the objective of minimizing the monetary cost [22], the makespan [9], [29] or both metrics [12]. However, most of these efforts target a specific framework (*e.g.*, Hadoop) so the proposed prediction models cannot be applied to others (*e.g.*, Spark). Additionally, they assume that the complete workload is known beforehand and do not consider the number of resources that should be reserved and how the applications' configuration parameters (*e.g.*, whether compression should be applied) affect their execution times.

Furthermore, there exist multiple cluster-wide schedulers like Apache Mesos [30] and Apache YARN [31] which enable the scheduling of big applications on shared resources. In contrast to those solutions, in our setup we have a dedicated cluster per framework and need to determine both the cluster and necessary resources and configurations for satisfying deadline constraints. A similar problem to the one we examine in this work, is the resource management and scheduling in multiple MapReduce clusters [11], [12]. Our setting is coherently different as we consider the fact that the performance of an application varies across different frameworks as well as the impact of various configuration parameters.

Moreover, there is increasing interest within the research community in the problem of modelling the impact of the amount of computational resources and the different framework's configuration parameters on the applications' execution time [18], [32]. Our work differs in that we consider how the implementation of an application on various big data frameworks can affect the performance and also extend a novel prediction technique based on decision trees [33], [34] to efficiently search the parameters' space and tune them appropriately so that applications can meet user defined deadlines. Finally, there are multiple auto-tuning frameworks [35] for configuring the parameters of big data frameworks but those are not generic and target specific frameworks [36].

VII. Conclusions

In this work we present *Orion* a novel online resource negotiator for scheduling big data applications across different big data frameworks. *Orion* exploits algorithmic improvements over the decision trees prediction technique to estimate the impact of the per framework configuration parameters on the application's execution time and also to efficiently traverse the large parameters' search space to determine the parameters' values that satisfy the application's performance requirements. Furthermore, *Orion* aims at minimizing the amount of resources that will be reserved by an application in order to reduce the user's budget cost without violating the applications' deadlines. Therefore, *Orion* automatically determines the framework to which each incoming application should be assigned and when execution of the application should start. It also appropriately tunes its configuration parameters. *Orion* can support different big data frameworks including but not limited to Apache Spark, Apache Flink and TensorFlow. Our experimental evaluation considering three state-of-the-art big data frameworks illustrates that *Orion* is capable of accurately estimating execution times, successfully meeting the deadline constraints with a high probability and minimizing the cost on the user's spending budget. In the future we plan to enable the preemptive scheduling of the assigned applications to be able to satisfy more deadlines.

Acknowledgment

This research has been financed by the European Union through the FP7 ERC IDEAS 308019 NGHCS project and the Horizon2020 688380 VaVeL and 690588 SELIS projects.

References

[1] N. Zacheilas, V. Kalogeraki, N. Zygouras, N. Panagiotou, and D. Gunopulos, "Elastic complex event processing exploiting prediction," in *BigData*. Santa Clara, CA, USA: IEEE, 2015, pp. 213–222.

[2] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White, "Towards expressive publish/subscribe systems," in *EDBT*. Munich, Germany: Springer, 2006, pp. 627–644.

[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[4] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *IEEE TCDE*, vol. 36, no. 4, 2015.

[5] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning." in *OSDI*, vol. 16, Savannah, GA, USA, 2016, pp. 265–283.

[6] "HDInsightHadoop, Spark, and R Solutions for the Cloud | Microsoft Azure." [Online]. Available: https://azure.microsoft.com/en-us/services/hdinsight/

[7] "Cloud Dataproc - Cloud-native Hadoop & Spark." [Online]. Available: https://cloud.google.com/dataproc/

[8] J. Li, J. Cheng, Y. Zhao, F. Yang, Y. Huang, H. Chen, and R. Zhao, "A comparison of general-purpose distributed systems for data processing," in *BigData, Washington, DC, USA*. IEEE, 2016, pp. 378–383.

[9] A. Verma, L. Cherkasova, and R. H. Campbell, "Orchestrating an ensemble of mapreduce jobs for minimizing their makespan," *IEEE TDSC*, vol. 10, no. 5, pp. 314–327, 2013.

[10] N. Zacheilas and V. Kalogeraki, "Real-time scheduling of skewed mapreduce jobs in heterogeneous environments," in *ICAC, Philadelphia, PA, USA*. Usenix, 2014, pp. 189–200.

[11] A. Iordache, C. Morin, N. Parlavantzas, E. Feller, and P. Riteau, "Resilin: Elastic mapreduce over multiple clouds," in *CCGrid, Delft, Netherlands*, 2013, pp. 261–268.

[12] N. Zacheilas and V. Kalogeraki, "Chess: Cost-effective scheduling across multiple heterogeneous mapreduce clusters," in *ICAC*. Würtzburg, Germany: IEEE, 2016, pp. 65–74.

[13] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *SIGMOD, Melbourne, VIC, Australia*. ACM, 2015, pp. 239–250.

[14] A. Gounaris, G. Kougka, R. Tous, C. Tripiana, and J. Torres, "Dynamic configuration of partitioning in spark applications," *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[15] Apache Beam, https://beam.apache.org/, 2018.

[16] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and regression trees*. CRC press, 1984.

[17] A. Karalič, "Employing linear regression in regression tree leaves," in *ECAI, Hamburg, Germany*, 1992, pp. 440–441.

[18] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: efficient performance prediction for large-scale advanced analytics," in *NSDI*. Santa Clara, CA, USA: Usenix, 2016, pp. 363–378.

[19] T. Achterberg, "Constraint integer programming," 2007.

[20] ——, "Scip: solving constraint integer programs," *Mathematical Programming Computation*, vol. 1, no. 1, pp. 1–41, 2009.

[21] A. H. Land and A. G. Doig, "An automatic method of solving discrete programming problems," *Econometrica: Journal of the Econometric Society*, pp. 497–520, 1960.

[22] Y. Wang and W. Shi, "Budget-Driven Scheduling Algorithms for Batches of MapReduce Jobs in Heterogeneous Clouds," *Cloud Computing, IEEE Transactions*, 2014.

[23] T. Bertin-Mahieux, D. P. Ellis, B. Whitman, and P. Lamere, "The million song dataset," in *ISMIR, Miami, FL, USA*, 2011.

[24] MovieLens, https://grouplens.org/datasets/movielens/.

[25] K-Means Data Generator, https://github.com/apache/flink/blob/master/flink-examples/flink-examples-batch/src/main/java/org/apache/flink/examples/java/clustering/util/KMeansDataGenerator.java.

[26] M. Poess and C. Floyd, "New tpc benchmarks for decision support and web commerce," *SIGMOD, Dallas, TX, USA*, vol. 29, no. 4, pp. 64–71, 2000.

[27] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[28] S. Maroulis, N. Zacheilas, and V. Kalogeraki, "Express: Energy efficient scheduling of mixed stream and batch processing workloads," in *ICAC*. Colombus, OH, USA: IEEE, 2017, pp. 27–32.

[29] Y. Wang, J. Tan, W. Yu, L. Zhang, X. Meng, and X. Li, "Preemptive reduce task scheduling for fair and fast job completion." in *ICAC*, San Jose, CA, USA, 2013.

[30] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI, Boston, MA, USA*, 2011.

[31] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *SOCC*. New York, NY, USA: ACM, 2013, pp. 5:1–5:16.

[32] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *NSDI, Boston, MA, USA*, 2017, pp. 469–482.

[33] K. Lolos, I. Konstantinou, V. Kantere, and N. Koziris, "Adaptive state space partitioning of markov decision processes for elastic resource management," in *ICDE, San Diego, CA, USA*. IEEE, 2017, pp. 191–194.

[34] ——, "Elastic management of cloud applications using adaptive reinforcement learning," in *BigData*. Boston, MA, USA: IEEE, 2017.

[35] V. Dalibard, M. Schaarschmidt, and E. Yoneki, "Boat: Building autotuners with structured bayesian optimization," in *WWW, Perth, Australia*, 2017, pp. 479–488.

[36] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, "Starfish: A self-tuning system for big data analytics." in *CIDR*, vol. 11, 2011, pp. 261–272.