Noname manuscript No.
(will be inserted by the editor)

Predicting the Performance of Big Data Applications on the Cloud

D. Ardagna · E. Barbierato · E. Gianniti · M. Gribaudo · T. B. M. Pinto · A. P. C. da Silva · J. M. Almeida

the date of receipt and acceptance should be inserted later

Abstract Big data analytics have become widespread as a means to extract knowledge from large datasets. Such applications are often characterized by highly heterogeneous and irregular data access patterns, challenging existing software and hardware infrastructures to meet their dynamic resource demands. The cloud computing paradigm, in turn, offers a natural hosting solution to such applications as it provides flexibility and elasticity, adapting the allocated resources in response to the application's current needs. However, these properties impose extra challenge to the accurate performance prediction of cloud-based applications, which is a key step to adequate capacity planning and managing of the hosting infrastructure. In this article, we tackle this challenge by exploring three modeling approaches for predicting the performance of big data applications running on the cloud. We evaluate two queuing-based analytical models and a novel fast ad-hoc simulator in various scenarios based on different applications and infrastructure setups. The considered approaches are compared in terms of prediction accuracy and execution time. Our results indicate that our two best approaches can predict average application execution times with only up to a 7% relative error, on average. Moreover, both of them run very fast (requiring at least two orders of magnitude lower execution time than widely used tools while providing slightly better accuracy), being practical for online prediction.

Keywords Performance Prediction · Apache Spark · Parallel Computing · Big Data · Analytical and Simulation Models

D. Ardagna, E. Barbierato, E. Gianniti, M. Gribaudo
Dipartimento di Elettronica, Informazione e Bioingegneria,Politecnico de Milano, Italy
emaildanilo.ardagna@polimi.it, enrico.barbierato@polimi.it, eugenio.gianniti@polimi.it, marco.gribaudo@polimi.it

T. B. M. Pinto, A. P. C. da Silva, J. M. Almeida
Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Brazil
emailtuliobraga@dcc.ufmg.br, ana.coutosilva@dcc.ufmg.br, jussara@dcc.ufmg.br

1 Introduction

Big data analytics have become widespread as a means to extract knowledge from large datasets. Such applications have moved from experimental setups to enterprise-wide deployments bringing innovation and competitive advantage to many businesses [**?**]. Indeed, it has been reported that the big data market increased from $ 3.2 billion in 2010 to almost $ 17 billion in 2015 [1].

Besides the high volumes of data, big data applications are often characterized by increasing heterogeneity and irregularity in data access patterns. Such properties impose challenges to the hardware and software hosting infrastructure. In turn, cloud computing infrastructures have become a versatile computing platform as cloud resources fit the application requirements as they are needed, leveraging the elastic nature of the cloud. Thus, big data applications find in cloud-based infrastructures a natural hosting platform to cost-effectively provision the resources necessary for their execution. Indeed, in 2016 61% of applications that adopted Spark,[2] a fast and general engine for large-scale data processing, ran on the cloud [3].

Though flexible, the shared infrastructure that powers the cloud, when coupled with the natural irregularity of big data applications, offers extra challenges to the performance prediction of such cloud-based big data applications. Yet, accurate performance prediction is a key step for the planning and managing of any system, as it drives the automatic system (re-)configuration so as to meet applications' dynamic needs, avoiding Service Level Agreement (SLA) violations.

There is a rich body of work on performance modeling techniques, varying from analytical approaches to simulation tools [**?**, **?**, **?**, **?**, **?**, **?**, **?**]. However, their efficiency in modeling massively parallel applications by introducing thousands of parallel tasks has been shown to be an issue [**?**]. Thus, here we take the challenge of predicting the performance of cloud-based big data applications by exploring three very different techniques, two analytical models and a simulation tool, which, as will be discussed, have complementary pros and cons in terms of prediction accuracy and time efficiency.

Our present goal is to efficiently estimate (in a few seconds), the *average execution time* of a target application, given the available resources, in a way we can support run-time reconfiguration decisions. That is, given a target application, specified by a directed acyclic graph (DAG) representing the individual tasks and their parallelism and dependencies, the purpose is to predict how long it will take for the application to run (on average) on a given resource deployment (described in terms, e.g., of numbers of cores or nodes) by relying on hystorical data. We focus on applications running on Spark, whose adoption has steadily increased and which probably will be the reference big data engine for the next 5–10 years [4].

[1] http://idcdocserv.com/1414

[2] http://spark.apache.org/

[3] https://databricks.com/blog/2016/09/27/spark-survey-2016-released.html

[4] http://fortune.com/2015/09/25/apache-spark-survey

Firstly, we explore a technique based on a simple upper-bound on the average execution time for Fork-Join queuing networks, proposed by Nelson and Tantawi [**?**], which depends only on the number of parallel tasks and on the average execution time of a single task. We refer to this model as *Fork-Join model*. As an alternative, we also investigate the use of a more sophisticated analytical queuing network (QN) model, which was originally proposed in [**?**] for performance prediction of parallel application. The QN model extends an Approximated Mean Value Analysis (AMVA) technique by modeling the precedence relationships and parallelism between individual tasks of the same job. This model, here referred to as *Task Precedence model*, explicitly captures the overlap in execution times of different tasks of the same job to estimate the average application execution time. We do expect that the *Task Precedence model* outperforms the simpler *Fork-Join* approach. However, our aim in considering the latter in our evaluation is to assess the extent to which such simpler approach is able to capture major components of the performance of the target applications.

We also propose and evaluate *dagSim*, a novel ad-hoc and fast discrete event simulator to model the execution of complex DAGs. Compared to other formalisms (e.g., Stochastic Petri Nets) or specific tools (e.g., JMT [**?**] and GreatSPN [**?**]), we find that the dagSim simulation process achieves similar or better accuracy within a much shorter timescale (up to two orders of magnitude faster).

We evaluate the modeling approaches in seven scenarios consisting of different virtual machine environments and applications, as well as different resource configurations. Our experimental results indicate that the simple Fork-Join model is too simplistic for the target scenarios, providing very inaccurate results. On the other hand, we find a good overall accuracy for both the Task Precedence model and dagSim simulator, with average relative errors, across all considered scenarios and configurations, of only 7.38% and 5.65%, respectively. Specifically dagSim performed better for interactive queries while the Task Precedence model performed better for iterative machine learning (ML) algorithms.

We also evaluate the two most promising prediction models regarding their execution times. Our results indicate that, being an analytical tool, Task Precedence runs faster, although both models have execution times that are suitable for online prediction. Morever, dagSim has the extra advantage of providing not only averages, but also percentile estimates of the application execution times.

This work builds on our prior preliminary effort [**?**] by (1) considering a third modeling approach, the simpler Fork-Join model to assess the extent to which it still can provide cost-effective results; (2) including three new scenarios in our evaluation; (3) deepening the analysis of dagSim while providing quantile results and an extended comparison against JMT.

The rest of this paper is organized as follows. Section 2 presents related work, while Section 3 introduces our prediction models. Section 4 describes the

experimental scenarios we explored and discusses our main results. Section 5 offers conclusions and possible future work.

2 Related Work

The performance analysis and prediction of big data applications running on the cloud can be tackled from different perspectives. Indeed, a number of sophisticated projects focusing on the performance of Spark applications have recently emerged. For example, PREDIct [**?**] includes a set of prediction techniques for different areas of data analytics, while RISE2016 [**?**] is a collection of scalable performance prediction techniques for big data processing in distributed multi-core systems. In [**?**], in turn, the authors provide hierarchical models that leverage the multi-stage execution structure of Spark jobs. They are able to obtain good accuracy generalizing the measurements performed on a fraction of the real application data set. In the following, we focus on more general solutions that exploit either (i) analytical queuing network models or (ii) simulation approaches.

2.1 Analytical Queuing Network Models

Applications running in parallel systems have to share physical resources (processors, memory, bus, etc.). Competition for computational resources can occur among different applications (inter-application concurrency) or among tasks of the same application (intra-application concurrency). Given system resource limitations, performance analysis techniques are important for studying fundamental performance measures, such as mean response time, system throughput, and resource utilization. In this context, queuing network (QN) models have been successfully used for studying the impacts of resource contention and queuing for service in the applications running on top of parallel systems [**?**, **?**, **?**, **?**].

The parallel execution of multiple tasks within higher level jobs is usually modeled in the QN literature with the concept of fork/join: jobs are spawned at a fork node in multiple tasks, which are then submitted to queuing stations modeling the available servers. After all the tasks have been served, they synchronize at a join node. Unfortunately, there is no known closed-form solution for fork-join networks with more than two queues, unless a special structure exists [**?**].

The authors in [**?**] present a model for predicting the response time of homogeneous fork/join queuing systems. The observed system is made up of a cluster of *homogeneous* index servers, each holding portions of queriable data, and the query requests to the index servers go in an FCFS (First-Come First-Served) scheduling queuing discipline. In order to represent system parallelism, the index server subsystem is modeled as a fork-join network. In this model, an incoming task is split (forked) into identical subtasks, which are then sent to

individual servers and executed in parallel, independently from one another. Once all subtasks have finished executing, they are joined and the task execution is completed. The average response time is determined by the slowest server.

Following the fork-join model paradigm, the authors in [**?**] present an analysis of closed, balanced fork-join queuing networks, in which a fixed number of identical jobs circulate. They introduce an inexpensive bounding technique referred to as balanced job bounds for fork-join systems (BJB-FJ), which is analogous to balanced job bounds developed for product form networks. Servers have an FCFS queuing discipline and exponentially distributed service times. Based on Markov models theory, the authors provide accurate approximation results for the job response time. Unfortunately these results cannot be adopted to big data systems where jobs are not balanced and tasks execution time follows different distributions [**?**]. In the same direction, the authors in [**?**] model a multiprocessing computer system as K homogeneous servers, each with an infinite capacity queue. Jobs arriving at the system are split into K independent tasks, each of which is assigned to a server. The authors provide a computationally efficient algorithm for obtaining upper and lower bounds on the expected response time of this system. Moreover, the algorithm guarantees an error bound and, if one desires, tighter error bounds can be obtained at the cost of more computation.

The work in [**?**] also considers the issue of estimating performance metrics in parallel applications. The proposed method is computationally efficient and accurate for predicting performance of a class of parallel computations, which can be modeled as task systems with deterministic precedence relationships represented as series-parallel DAGs. Tasks are represented as nodes and edges mark precedence relationships between pairs of nodes. A task can be executed once its parent tasks have finished executing. Furthermore, whenever nodes are independent, their executions may overlap fully or partially, according to resource availability. This overlap can be determined from the start and end times of task executions. The amount of overlap between tasks can then be used to reduce the initial task DAG and successively estimate task response times, ultimately leading to an estimate of the full application response time. While the models proposed in [**?**, **?**, **?**] assume a fork-join abstraction to represent parallel behavior, here the authors focus on the precedence relationships resulting from tasks that must run sequentially, combined with those that may run in parallel. An extension of this model, capturing not only intra-job, but also inter-job execution overlap to evaluate application response times, is presented in [**?**].

In our work, we apply the models proposed by the authors of [**?**] and [**?**], given that the parameters of both models (for instance, service demands and task structure) can be easily obtained, and results are obtained with low complexity cost. More details on [**?**] and [**?**] models are presented in Sections 3.2.1 and 3.2.2.

2.2 Simulation Approaches

Several simulation tools, which are tailored to study the behavior of parallel applications through stochastic formalisms such as Stochastic Petri Nets (SPNs) [**?**], have been implemented. Some examples include the Stochastic Petri Net Package (SPNP) [**?**] and GreatSPN [**?**]. In particular, GreatSPN supports the analysis of Generalized Stochastic Petri Nets (GSPNs) including both immediate and timed (the fire event occurs either immediately or within a stochastic time) transitions and of Stochastic Well-Formed Nets (SWNs, i.e., Petri nets where tokens can be distinguished) [**?**]. Also, SMART (Symbolic Model checking Analyzer for Reliability and Timing) [**?**] includes both stochastic models and logical analysis, whereas SHARPE (Symbolic Hierarchical Automated Reliability and Performance Evaluator) [**?**] is a tool to analyze stochastic models, the most notable being fault trees, product form queuing networks, Markov chains, and GSPNs. JMT [**?**], in turn, is a suite of applications offering a framework for performance evaluation, system modeling, and capacity planning.

The problem of studying the performance prediction of individual jobs is explored in [**?**] through a framework consisting of a Hadoop job analyzer, while the prediction component exploits locally weighted regression methods. A similar issue is studied in [**?**] by using instead a hierarchical model including a precedence graph model and a queuing network model to simulate the intra-job synchronization constraints. In [**?**], the authors consider the problem of minimizing the cost involved in the search of the optimal resource provisioning, proposing a cost function that takes into account the time cost, the amount of input data, the available system resources (Map and Reduce slots), and the complexity of the Reduce function for the target MapReduce job. The usage of a simulator to better understand the performance of MapReduce setups is described in [**?**] with particular attention to the effect of several component inter-connect topologies, data locality, and software and hardware failures.

Our previous work [**?**] describes multiple queuing network models (simulated with JMT) and stochastic well formed nets (simulated with GreatSPN) to model MapReduce applications, highlighting the trade-offs and additional complexity required to capture system behavior and improve prediction accuracy. Simulation times demonstrated to be prohibitive (from several minutes to hours for the more general cases). As a result, general purpose simulators such as GreatSPN and JMT are not suitable to efficiently study massively parallel applications introducing tens (or even hundreds) of stages and thousands of parallel tasks for each stage.

Finally, parallel and distributed processing have been investigated also by means of Process Algebra (PA) [**?**]. A PA is a mathematical framework describing how a system evolves by using algebraic components and providing a set of methods for their manipulation. Among the different implementations, Performance Evaluation Process Algebra (PEPA) [**?**] is a formal language for distributed systems, whose models correspond to continuous time Markov chains (CTMC).

We here propose a novel and fast discrete event simulation tool, called dagSim, which was built to analyze DAG-based applications. Compared to previous simulation approaches, our tool achieves greater prediction accuracy within shorter timescales. Indeed, results comparing dagSim against JMT are shown in Section 4.2.

3 Performance Prediction Models

This section presents the three modeling approaches analyzed in this paper to predict the performance of cloud-based big data applications, namely Fork-Join, Task Precedence, and dagSim. Since our focus is on applications running on Spark, we start by first presenting some key components of this framework in Section 3.1, highlighting assumptions behind its parallel execution model that influence performance modeling. The two analytical queuing network models and the proposed dagSim simulator are then described in Sections 3.2 and 3.3, respectively. Finally, Section 3.4 overviews the architecture of the performance tools we developed to automate the performance analysis of Spark applications from low level logs.

3.1 Spark Overview and Model Assumptions

Spark is a fault-tolerant cluster computing framework that provides a set of abstractions for parallel computation across distributed nodes with multiple cores. It is a fast and general purpose engine for large-scale data processing and it was first proposed as an alternative to Hadoop MapReduce [**?**]. Spark is the state-of-art for fault-tolerant parallel processing and it recently became popular for big data processing on the cloud [**?**].

The general unit of computation in Spark is an application. It may be composed of a single job, multiple jobs, or continuous processing. Jobs are sequentially executed by default. A job, in turn, is composed of a set of data transformations and terminates with an action requesting a value from the transformed data. Each transformation represents a specific piece of code that launches data-parallel tasks on read-only data divided into blocks of almost equal size, called partitions. This set of same class tasks is called stage. Within a stage, a single task is launched for each data partition, thus the number of tasks inside the stage is equal to the number of partitions. During the stage run time, each core (also called CPU slot) can run only a single task at a time. Since cores are a limited resource, the tasks are assigned to CPU slots until all resources become busy. Thus, the remaining tasks are enqueued and scheduled to be executed as soon as the cores become available.

The Spark execution model is represented by a DAG. Considering a logical plan of transformations that is fired by an action, the Spark *DAGScheduler* constructs a DAG of stages and their precedence relations. The stages are submitted for execution as a set of tasks that follows FCFS policy. The

TaskScheduler does not know the dependencies between stages. Each stage is a fully-independent sequence of tasks that can run right away based on the data already on the cluster [**?**]. Thus, only stages have precedence relationships and these are represented by the DAG.

Our present goal is to apply a set of performance prediction techniques to estimate the execution time of Spark applications and evaluate their effectiveness. The issue of performance prediction in parallel systems has been approached in several ways, with varying degrees of detail, cost, and accuracy. Focusing on such data-parallel frameworks based on a DAG execution model, one of the main concerns is to model the synchronization step that happens when a stage terminates. That is, models to predict application performance must take into account how the executions of stages overlap among themselves.

In this work, we made the following assumptions for all the three (analytical and simulation) performance models: i) the concurrent system is modeled as a closed queuing model, with a single application that splits into one or more Spark jobs, ii) jobs are sequentially scheduled and consist of one or more stages, iii) multiple stages may run in parallel or may have some precedence relationships, iv) a stage is composed of tasks of the same class with no precedence relationship among themselves (i.e., they may run in parallel), v) an individual application obtains dedicated resources for its execution (i.e., VMs that are executed on a cloud cluster), vi) resources (such as memory, CPU, disk) are homogeneous (as frequently happens in cloud deployments, see, e.g., [**?**]).

Moreover, the two queuing network models are based also on the assumption that the times required to process each block of computation are exponentially distributed [**?**, **?**]. For the Fork-Join model, the application execution is modeled as a sequence of one or more fork-joins, as will be described below, and a block of computation is a node in each fork-join. For the Task Precedence model, each block of computation is a Spark stage. Clearly, in both cases, this assumption may not hold in practice, possibly depending on characteristics of the application. In other words, it is a potential source of approximation error of the models. Assessing the extent to which such approximation errors become prohibitively large is part of our goal.

3.2 Analytical Queuing Network Models

This section describes the two analytical queuing network models explored in this paper. We first present the Fork-Join model, which relies on a known approximation of response time for parallel applications and then briefly describe the Task Precedence model, which takes as input a DAG representation of the parallel application.

3.2.1 Fork-Join Model

This model [**?**] provides a very simple upper bound on the average execution time for queuing networks with fork-join synchronization. In general, the main idea is as follows: the cluster architecture is represented as a two-level graph in which the leaves represent the execution nodes, modeled as Spark worker cores, and the root represents the unit that controls the execution flow, modeled as the Spark master node. Stages are forked into same class tasks and scheduled to execute in parallel across the available cores. After execution at all worker cores, the results from each unit are joined.

Specifically, the application execution is broken into sequential *phases*, each one represented by a fork-join structure. Each such structure captures the execution of a single Spark stage or multiple stages that run concurrently (or partially concurrently and partially sequentially). Thus, this is a coarser representation of the application DAG where multiple nodes may be merged into a single one. We use the model to estimate the execution time of each fork-join structure separately, as described below, and predict the total application execution time as the sum of the execution times of all fork-joins.

For each fork-join structure, the model expects as input the number of execution nodes, i.e., the total number of cores available across the worker nodes, and the average execution time of that block of computation at an individual core. The execution time at a single core can be estimated based on historical data (i.e., logs of previous executions of the same Spark application). The output consists of two values, namely a lower and an upper bound. The lower bound is the execution time at a single core, here referred to as R_{core}, and reflects the execution time when there are no synchronization delays caused by discrepancies across individual cores. In the present case, this lower bound is not of interest, for prediction purposes, as it is one of the model inputs. However, R_{core} could be estimated based on some finer-grained modeling strategy (e.g., Mean Value Analysis). The upper bound, on the other hand, is our main interest: it provides an approximation that aims to capture the effects of a slower core, which would cause delays and affect the overall execution times.

According to Nelson and Tantawi [**?**], an upper bound for the execution time of a fork-join structure running on K cores is given by the product of the average execution time at a single core (R_{core}) and the H_K harmonic number (more details in [**?**]). Thus, the bounds for the execution time (R) are:

$$R_{\text{core}} \leq R \leq H_K R_{\text{core}},$$

with $H_K = 1 + 1/2 + 1/3 + \cdots + 1/K$.

We note that the sequence of fork-join structures used by this model may be a very coarse approximation for the execution of Spark applications. In particular, it does not explicitly capture all precedence relationships among Spark stages. Rather it assumes some of those relationships, particularly those among stages merged into a single fork-join, are indirectly captured in the input (R_{core}). In other words, by looking at historical data, we compute the

average total execution time at each individual core, considering all stages and individual tasks of the given block of computation. By doing so, we capture the precedence of individual tasks/stages in a single core. The model, in turn, aims at capturing all synchronization delays across different cores via the inflation factor H_K. Though a coarse approximation, the simplicity of this approach motivated us to assess to which extent it can provide reasonably accurate (from a practical perspective) performance predictions for Spark applications.

3.2.2 Task Precedence Model

In this prediction method, the performance of a parallel application is modeled by explicitly capturing the precedence relationships between different blocks of computation. We start by presenting the main ideas behind the model, as proposed in [**?**]. We refer to the original paper for a detailed derivation of the model. We then discuss how we applied this model to Spark applications at the end of the section.

In the original paper [**?**], each block of computation was called a task, and the goal was to estimate the average execution time of an application composed of multiple parallel/sequential tasks. The precedence relationships between different tasks are expressed as a series-parallel DAG, where each node is a task. Available resources (e.g., cores) are modeled as service centers in a queuing network model. By exploiting both the queuing network and the DAG, the authors modified a traditional iterative MVA approach to account for delays caused by synchronization and resource constraints originated from task precedence and parallelism.

The solution uses a traditional MVA model to estimate the average execution time of each task. In order to explicitly capture the synchronization delays between parallel tasks, the model estimates an *overlap* probability between each pair of tasks based on the input task precedence DAG. This probability captures the chance that the executions of the two tasks overlap in time, and is used as an inflation factor to estimate a new set of task average execution times, according to the MVA equations. The model will continue to iterate over these values until they converge below a given error threshold. As a final step of each iteration, the precedence graph is reduced to determine the average execution time of the whole application. For example, execution times of sequential tasks will be added; execution times of parallel applications will be aggregated according to a probabilistic approach that takes into account the overlap probabilities between them.

Since jobs in Spark are sequentially executed by default, we here apply the model by considering each node in the input DAG as a stage of the Spark application, thus explicitly capturing the dependencies among stages (unlike the Fork-Join model) that exist in Spark applications. Each stage is fully described by its average execution time,[5] which is estimated based on historic

[5] As mentioned, the model assumes that the execution times of the nodes in the input DAG, i.e., the stages of the Spark application, are exponentially distributed.

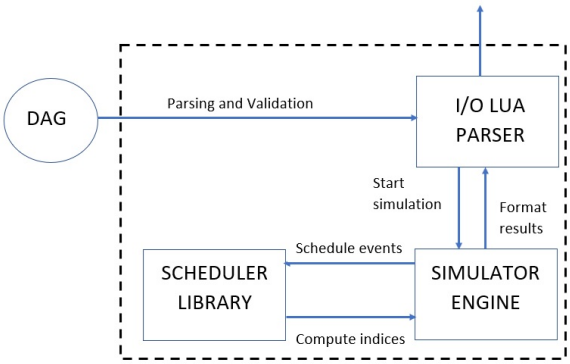

Figure 1 dagSim and its components

data (Spark logs of previous executions of the same application). Thus, the model takes as input the application's DAG and the average execution time of each individual stage and outputs the average execution time of each job. To estimate the average execution time of the application, the execution times of all jobs are simply added together.

3.3 The discrete event simulator

dagSim (see Figure 1) is a high speed discrete event simulator built to analyze DAG-based jobs,[6] and it consists of three main components: i) a *parser*, ii) a *simulation component*, and iii) an *output module*.

Models are described with a data driven approach defining the DAG stages and the workload they have to handle. Specifically, a DAG model is defined as a tuple:

$$DAG = (S, N_{\text{Cores}}, N_{\text{Users}}, \mathcal{Z}) \tag{1}$$

Where $N_{\text{Cores}} \in \mathbb{N}, N_{\text{Cores}} \geq 1$ represents the number of computational cores and $N_{\text{Users}} \in \mathbb{N}, N_{\text{Users}} \geq 1$ the number of users concurrently submitting Spark applications to the system, and $\mathcal{Z}$ is the *think time distribution*: the time a user will wait before submitting a new application. $S = \{s_1, \ldots, s_{N_{\text{Stages}}}\}$ is the set of stages that define the DAG. Furthermore, each stage $s_i \in S$ is a tuple:

$$s_i = (\texttt{id}, N_{\text{Task}}, Pre, Post, \mathcal{T}) \tag{2}$$

where `id` is a symbolic constant assigning a name to the stage, $N_{\text{Task}} \in \mathbb{N}, N_{\text{Task}} \geq 1$ accounts for the tasks composing the stage, $Pre \in S$ and $Post \in S$ define respectively the stages that must have been completed for s_i to be executable, and the set of stages that will be able to run after the completion of s_i. The probability distribution $\mathcal{T}$ defines the duration of each task of the stage.

[6] The tool is available at `https://github.com/eubr-bigsea/dagSim`

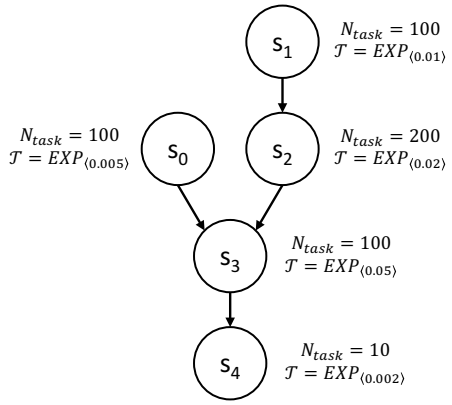

Figure 2 An example of a Spark DAG

For example, the DAG presented in Figure 2 describes a Spark application composed of five stages, s_0–s_4. All tasks are characterized by an exponentially distributed random duration, whose rate depends on the stage. s_0 and s_1 can start immediately, while s_2 requires the completion of s_1 to start. Stage s_3 can be executed only after s_0 and s_2 completed, and s_4 must be performed after s_3. In this case we have the stages defined by $S = \{s_0, s_1, s_2, s_3, s_4\}$ with:

$$
\begin{aligned}
s_0 &= (\mathsf{s}_0, 100, \emptyset, \{s_3\}, EXP_{<0.005>}) \\
s_1 &= (\mathsf{s}_1, 100, \emptyset, \{s_2\}, EXP_{<0.01>}) \\
s_2 &= (\mathsf{s}_2, 200, \{s_1\}, \{s_3\}, EXP_{<0.02>}) \\
s_3 &= (\mathsf{s}_3, 100, \{s_0, s_2\}, \{s_4\}, EXP_{<0.05>}) \\
s_4 &= (\mathsf{s}_4, 10, \{s_3\}, \emptyset, EXP_{<0.002>})
\end{aligned}
$$

If we consider that the application is executed by $N_{\text{Users}} = 10$, each one characterized by an exponentially distributed think time with an average of 1000 s, on a cluster composed by $N_{\text{Cores}} = 64$ cores, we then have:

$$DAG = (\{s_0, s_1, s_2, s_3, s_4\}, 64, 10, EXP_{<0.001>}) \tag{3}$$

As a file interchange format to encode the previously formalized models (for example Equation 3), Lua, a multi-purpose procedural programming language [7], was selected. There are several advantages in using Lua with respect to other alternatives. First, it has a very compact syntax for defining complexly structured constants, which allows initializing complex objects with a short textual overhead. Other languages, such as XML, require a complex markup structure, which in the end makes input files very long and difficult to

[7] https://www.lua.org/home.html

generate. In our approach, an input model is an instantiation of a set of predefined global variables: thanks to the names used to identify such variables, files are easily readable by a human, and simple to automatically generate by a software component. Next, even if only the instances of the global variables are used, model files are effectively Lua programs. This means that they can exploit a full set of instructions and commands to algorithmically compose a model by loading pieces from external datasets, computing rates with algebraic expressions (e.g., 1/10000 instead of 0.0001), and use loop constructs to repeat the same assignment several times.

For example, the DAG shown in Figure 2 and described in Equation 3 is encoded with the following Lua code:

```
Stages = {
  { name = "S0", tasks = 100, pre = {}, post = {"S3"},
    distr = {type = "exp", params = {rate = 1/200.0}} },
  { name = "S1", tasks = 100, pre = {}, post = {"S2"},
    distr = {type = "exp", params = {rate = 1/100.0}} },
  { name = "S2", tasks = 200, pre = {"S1"},
    post = {"S3"},
    distr = {type = "exp", params = {rate = 1/50.0}} },
  { name = "S3", tasks = 100, pre = {"S0","S2"},
    post = {"S4"},
    distr = {type = "exp", params = {rate = 1/20.0}} },
  { name = "S4", tasks = 10, pre = {"S3"}, post = {},
    distr = {type = "exp", params = {rate = 1/500.0}} }
};
Cores = 64;
Users = 10;
UThinkTimeDistr = {type = "exp",
                   params = {rate = 1/1000.0}};
```

The performance indices computed during the simulation are fed back into Lua variables. The simulator then executes some predefined Lua code that outputs the results stored in such variables. Currently, performance indices are displayed as plain text. However, the ability of the user to replace the legacy Lua output code with custom procedures allows to easily integrate the tool within larger frameworks. Further enhancements may include library procedures to support different formats, such as CSV, XML, or HTML.

The simulation engine has been written in the C language. It is based on a classic discrete event simulation algorithm and has been designed for high performance. Though dagSim is a lightweight tool compared to other commercial programs, it targets specifically DAG models. Simulation can run efficiently thanks to a proprietary scheduler library we developed offering data structures that perform well when a high volume of events is generated. The tool is highly portable, since it can be easily recompiled without the requirement of external tools or libraries not supplied with the source code.

In the following, we will use the following notations and definition to describe the simulation procedure:

- Each *stage* s_i consists of one or more *tasks* $\{t_1, t_2, \ldots, t_m\}$;
- Each *task* corresponds to an event and is characterized by a *timestamp*, denoting a start time;

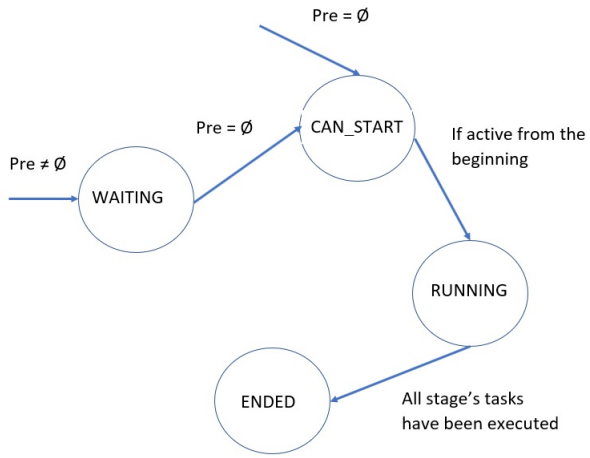

Figure 3 Stage finite state machine

- Given a stage s_i, Pre_i denotes the set of stages that need to be finished for s_i to start.
- According to the finite state machine depicted in Figure 3, a stage s_i of a Spark application can be in one of the following four states $ST(s_i)$:
  - CAN_START: $Pre_i = \emptyset$ or $Pre_i \neq \emptyset$ and all stages belonging to Pre_i are in state ENDED ($\forall s_j \in Pre_i : ST(s_j) = \text{ENDED}$).
  - WAITING: $Pre_i \neq \emptyset$, but some of the stages belonging to Pre_i are in state WAITING or RUNNING ($\exists s_j \in Pre_i : ST(s_j) = \text{WAITING} \vee ST(s_j) = \text{RUNNING}$)
  - RUNNING: tasks belonging to stage s_i are in execution;
  - ENDED: all the tasks in stage s_i have been completed.

Initially, only the stages s_i that have no dependencies (i.e., such that $Pre_i = \emptyset$) are in the CAN_START state, and all the other are in the WAITING state. Each stage in the RUNNING state exploits a variable to count the number of tasks that still need to be completed. The core idea of the simulation engine is that each time a stage $t_k \in s_i$ has been executed, a counter is decremented of one unit. When the counter reaches zero, the engine determines that a stage is completed and selects which ones are now eligible to start, changing their state from WAITING to CAN_START.

By using a doubly-linked list that stores the relevant information about the jobs whose stages are in the CAN_START state, it is possible to determine which one can be executed without performing a full search on the set. In this sense, the approach provided by dagSim's engine is original and more efficient with respect to other scheduling mechanisms implemented in JMT [**?**] or GreatSPN [**?**].

Algorithm 3.1 summarizes the procedure to simulate the execution of one application according to the given DAG and Table 1 presents the relevant

Table 1 Notable data structures

Data structure	**Relevant parameters**
Model	Definition of the DAG.
CalendarEvent	Data structure to contain the events of the simulation.
Users	Structure to contain performance indices for the users.
UserAppData (UAD)	- User identifier - Application identifier - Start and End time - Stages that still needs to be started - Applications that still needs to be completed - State of the stages - Start / End time for the stage - Lists of applications that can be started
CoreData	- Number of free cores - User locker - Applications that still needs to be executed
App	Linked list of AppData
AppData	- User identifier - Application identifier - Stage identifier - Task identifier
sList	- The description of an Event
sAuxlists	Auxiliary List to store Events that are in the waiting state

data structures. The procedure receives as parameters the model definition $Model\ M$, the data structures to collect the performance indices relevant to the considered users $Users\ U$, and a data structure to contain the event list $CalendarEvent\ ce$. Initially (lines 2–5), for each of the N_{Users} users accessing the system, a doubly-linked list called UAD is populated with a set of information, notably i) the number of stages ready to be started, ii) the remaining tasks that need to be completed for each stage, iii) the state of each stage, iv) the start and end time of each stage, and v) a pointer to a list of applications ready to start. The data structure modeling the execution cores is initialized at line 6: it is mainly used to determine whether a cores is free or working on a task. To simplify the presentation, the details on the implementation of these steps are not discussed: Table 2 summarizes the functions that perform secondary duties to support the simulation, together with a brief description of their goals.

The algorithm continues by scheduling the time at which each user submits her first application (lines 7–9) by adding a new event whose timestamp corresponds to the *think time* $\mathcal{Z}$. Events are collected in a *CalendarEvent* data structure using the *AddEvent* function, while *ThinkTime* returns an instance of distribution $\mathcal{Z}$. The data structure characterizing an application contains a

doubly-linked list *AppData*, populated by i) a user identifier, ii) an application and a stage status, and a iii) task identifier.

The most important part of the algorithm consists of the cycle repeating the simulation for all the considered applications (lines 11–37). At line 12, the next simulation event is extracted (*pop* operation) from the *CalendarEvent* structure. Depending on the type of the event, the simulator performs different steps. If the event represents a user requesting the launch of a new application (line 14), the function *initUserAppData* is invoked (line 15) to initialize all the application's stages to CAN_START or WAITING state depending on whether the stage has dependencies or not.

The simulator assumes that each application locks the cores until they are no longer needed since the number of remaining tasks is less than the acquired resources. This is implemented by exploiting a lock that is set when a new application starts and reset when all its stages have been started. If there are available computational cores and no lock has been set (line 16), the *scheduleReadyTasksOnAvailNodes* function is invoked (line 17) to i) set a lock if a new application is started and ii) schedule the waiting applications on available cores. If instead the application cannot be started, it is inserted into the auxiliary list (line 19) WAITLIST.

If the event identifies the end of a task (line 21), the corresponding counter of the remaining tasks in the stage is decremented by one unit (line 22).

The stage is considered to be over if there are no tasks left (line 23): in this case function *releaseCore* is invoked (line 24) in order to free the computational resources; this also removes the lock on the cores if the following conditions are met: i) no more tasks need to be executed, ii) no other user has locked the core, and iii) there are no other stages to start. The stage state is updated to ENDED (line 25) and the *updateStageStatus* function is invoked to see if the completion of this stage allows other stages to change their status from WAITING to CAN_START (line 26). If another stage can start (line 27), the new tasks are scheduled (line 28); otherwise the application is considered to be completed. The application ending time (line 30) is set at the current time and the next application from the same user is submitted after another think time (lines 31-32). To allow the simulation to stop when the total number of considered applications has been executed, the number of completed applications is increased (line 33).

3.4 Tools Architecture

To better support the performance evaluation of Spark applications, the input file for both dagSim and the modules that perform analytical approximations can be automatically generated starting from the logs of earlier runs. In particular, a Spark log analyzer has been created to allow the workflow shown in Figure 4. Starting from the experimental data collected after the execution of the considered Spark applications, the tool extracts their stage and task structure. It also determines the running time distribution of the tasks

Table 2 Notable functions

Function	Description
createUserAppData	create and populate a UsrAppData structure
initCoreData	Populates a CoreData structure
AddEvent	Adds a new event to Calendar Event data structure
ThinkTime	Returns an instance of the Think Time Distribution $\mathcal{Z}$
populateAppData	Initializes an AppData structure
maxApps	Returns the maximum applications number as per the Lua input file
pop	Performs a *pop* operation on a CalendarEvent data structure. The output is an Event (sList data structure)
initUserAppData	Populates AppData structure for a specific user
createReadyList	Populates ReadyList structure including the applications to be executed and update the application status
scheduleReadyTasks-OnAvailableCores	Schedules waiting applications on available cores
releaseCore	Free core's resources
updateStageStatus	If new stages can start, changes their state
newStageCanStart	Checks if new stages can start
setAppEndTime	Defines the application completion time
addToAux	Adds an Event to an Auxiliary data structure
isEmpty	Returns *true* of *false* depending on the list passed as argument is empty or not
getLock	Returns *true* or *false* depending on the application passed as argument has been locked by a user or not
setLock	Locks an application

for each stage, and the other important parameters to automatically analyze them in dagSim and in the analytical approximation tools. It then generates the Lua file for dagSim and a model parameter file for the analytical models. These files can later be exploited as a starting point to perform performance studies of the considered applications: this workflow results particularly useful since, in some cases, Spark applications can be composed of hundreds of stages, which would be otherwise impossible to be manually described. A noteworthy aspect is that the log parser can define models that either replay each task's duration exactly in the order observed on the real system, or that empirically fit the service time distribution by merging a number of distinct execution logs.

4 Experimental Results

In this section, we present the results of a set of experiments we performed to explore and validate both Fork-Join and Task Precedence analytical models as well as the dagSim simulator. Our experimental setup covers seven different scenarios defined based on different classes of applications and different virtual machine environments, though all of them were executed on the Microsoft Azure cloud platform. In the following, we first introduce the considered sce-

Algorithm 3.1 Simulation engine algorithm

```
1: function dagSim(Model M, Users U, CalendarEvent ce)
2:     UserAppData **UAD;
3:     for i ∈ Users do
4:         UAD[i] = createUserAppData(M);
5:     end for
6:     CoreData *CD = initCoreData(M);
7:     for i ∈ Users do
8:         nEv = AddEvent(ce, ThinkTime(M));
9:     end for
10:    int TotalAppEnded = 0;
11:    while TotalAppEnded < maxApps(M) do
12:        event = pop(CE);
13:        App *ad = event->data;
14:        if isNewAppStarting(event) then
15:            initUserAppData(ad->userId, M);
16:            if (CD->freeCores > 0) AND (!lock(CD)) then
17:                scheduleReadyTasksOnAvailCores(ce, CD);
18:            else
19:                addToAux(event, WAITLIST);
20:            end if
21:        else
22:            remainingTasksXStage[ad->stageId]–;
23:            if remainingTasksXStage[ad->stageId] ≤ 0 then
24:                releaseCore(currTime, ce, CD, UAD);
25:                setstatus(s_k, ENDED);
26:                updateStageStatus(UAD, M)
27:                if newStageCanStart(UAD, M) then;
28:                    scheduleReadyTasksOnAvailCores(ce, CD);
29:                else
30:                    setJobEndTime(currTime);
31:                    nEv = addEvent(ce, T);
32:                    nEv->data = populateAppData();
33:                    TotalAppEnded++;
34:                end if
35:            end if
36:        end if
37:    end while
38: end function
```

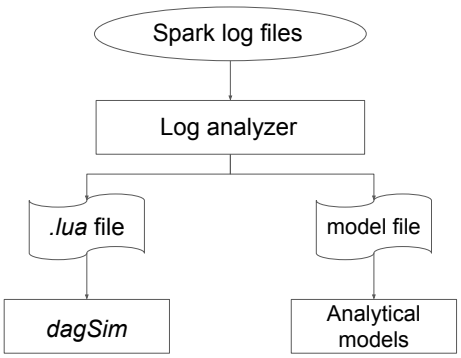

Figure 4 Integration workflow

Table 3 Experimental Deployment Configurations

VM	# cores	Executor Cores	Executor RAM	Driver RAM	VM RAM	VM Persistent disk
D12v2	4	2	2GB	4GB	28GB	200GB local SSD
A3	4	2	2GB	4GB	7GB	250GB local HDD
D4v2	8	4	10GB	8GB	28GB	400GB local SDD

narios and evaluation metrics, then discuss the results obtained on each of them.

4.1 Scenarios and Metrics

The experimental scenarios cover applications very widely used on Spark [**?**]. Specifically, we consider two SQL query workloads obtained from the TPC-DS industry benchmark[8] query execution plan, namely SQL queries 26 and 52 (Q26 and Q52, for short), as well as three reference machine learning (ML) benchmarks, namely K-means, Logistic Regression, and Support Vector Machine (SVM) [**?**]. The latter are iterative workloads that represent key steps in many ML applications and are becoming very popular in the Spark community [**?**]. The Q26 and Q52 workloads are examples of interactive queries that are often used as benchmarks for Spark platforms. Indeed nowadays big data applications are moving from the early days' batch processing to more interactive workloads.

We conduct our experiments on three types of virtual machine environments on the Microsoft Azure HDInsight PaaS [**?**], namely D12v2, A3 and D4v2, all of them running Spark. The goal is to explore different deployments of what the provider has to offer, including general purpose, CPU, and memory optimized instances. Considering that fault-tolerant parallel systems such as Spark are built to run on commodity clusters, it is important to guarantee the stability of the methods across different resource configurations.

Two different Spark versions have also been considered. The Spark 1.6.2 release and Ubuntu 14.04 were considered for the A3 and D12v2 VMs, while the D4v2 VMs feature Ubuntu 16.04 and Spark 2.1.0. All the scenarios have two dedicated master nodes over D12v2 VMs. Table 3 details the configurations, presenting for each type of VM the number of cores available per VM, number of cores and amount of RAM available for each workers' executor, amount of RAM for the master nodes' driver, as well as total RAM and persistent disk storage per VM.

For the A3 VMs, the workers' configuration varied from 6 up to 48 cores, while in the case of D12v2 VMs the number of cores varied between 12 and 52. The D4v2 deployments consisted of 24 and 48 cores, on three and six nodes respectively.

[8] http://www.tpc.org/tpcds/

Table 4 Scenarios Description

#	Application	VM	Configuration (nodes; cores; data)
1	TPC-DS Q26	D12v2	3–13; 4 cores per node; 500 GB
2	TPC-DS Q52	D12v2	3–13; 4 cores per node; 500 GB
3	TPC-DS Q26	A3	3–13; up to 4 cores per node; 500 GB
4	TPC-DS Q52	A3	3–13; up to 4 cores per node; 500 GB
5	K-Means	D4v2	3 and 6; 8 per node; 8 GB, 48 GB, 96 GB
6	Log. Regression	D4v2	3 and 6; 8 per node; 8 GB, 48 GB, 96 GB
7	SVM	D4v2	3 and 6; 8 per node; 8 GB, 48 GB, 96 GB

Table 4 describes the set of scenarios we analyze. Each TPC-DS query and machine learning benchmark was run 10 times for each considered configuration.

We evaluate all three prediction methods in terms of prediction accuracy and average execution time. The accuracy of each performance prediction model is estimated in each scenario by the relative error ε_{r}, computed using the average real execution time measured on the system T_{real} and the time predicted by the model T_{predict} for the considered application. That is:

$$\varepsilon_{\mathrm{r}} = \frac{T_{\mathrm{real}} - T_{\mathrm{predict}}}{T_{\mathrm{real}}}. \tag{4}$$

Note that a negative ε_{r} indicates that the model overestimates the application execution time, whereas a positive value implies the execution time was underestimated.

The execution times of all three models were measured on an Ubuntu 16.04 VirtualBox VM with 8 cores running on an Intel Nehalem dual socket quadcore system with 32 GB of RAM. The virtual machine has 8 dedicated physical cores with guaranteed performance and 4 GB of reserved memory. Unless otherwise noted, we report average execution times of 10 runs.

Before presenting the results for each scenario, we first compare the execution time of our novel dagSim simulator against that of the JMT [**?**], a widely used simulation tool.

4.2 dagSim versus JMT

In this section, we compare the average execution time of dagSim with that of the event based QN simulator available within the JMT 1.0.2 tool suite. JMT is very popular among researchers and practitioners and since 2006 has been downloaded more than 58,000 times. The comparison focuses on the average execution time at a 95% confidence level. The accuracy of JMT was analyzed and reported in our previous work [**?**], where we obtained an average percentage error of up to 33% while the mean of its absolute value was around 14.13%. The ratios between the average simulation times of JMT and dagSim for Scenarios 1 to 4 are reported in Figure 5. dagSim is clearly much faster

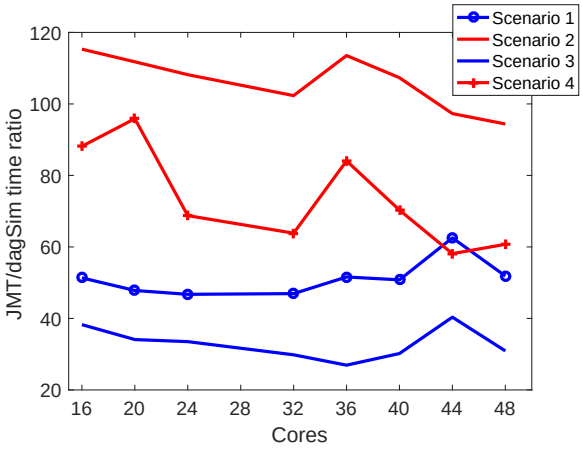

Figure 5 Ratio of JMT and dagSim execution times

than JMT (about 60 times on average across all runs and up to 115 times in the very worst case for the Q52 DAG in Scenario 2). We limit the comparison to TPC-DS queries, since ML workloads include a very large number of stages (more than 200 for SVM) and such large models cannot be built easily through JMT GUI. Moreover, as will be discussed extensively in the following sections, dagSim provides results that are slightly more accurate than JMT (as reported in [**?**]).

We present the results for each scenario, grouped by VM environment, in the following three sections.

4.3 Results on the D12v2 VMs (Scenarios 1 & 2)

This section presents results on the prediction accuracy of the Fork-Join [**?**] and Task Precedence [**?**] analytical models as well as the dagSim simulator over Spark 1.6.1 experiments executed on Azure HDInsight D12v12 VMs. Real and predicted application execution times for each scenario and various configurations (i.e., numbers of nodes and cores) are shown in Table 5. In this table, as in the following ones, relative errors of each model in each scenario/configuration are shown in parentheses, and maximum and minimum errors are shown in bold and shaded, respectively.

Considering scenario 1, the upper bound provided by the Fork-Join model has a relative error ε_r ranging from -256.18% to -162.27%. The maximum error was obtained for the largest configuration. We attribute these very large errors to the model's overestimation of delays caused by synchronization in our setup: as a single fork-join block of computation may be broken into successive steps, due to its resource demands and server capacity, extra synchronization overheads are introduced to the system and add to the overall response time of a single block. These response times are then overinflated by the model's use

Table 5 Scenarios 1 & 2: Real and predicted execution times (seconds) for Q26 and Q52 on D12v2 VMs. Results in *bold* for the maximum error and *shaded cells* for the minimum error.

Nodes (cores)	Real	Fork-Join (error %)	Task Prec. (error %)	DagSim (error %)
Scenario 1: TPC-DS Q26				
3(12)	722.2	1945.3 (-169.4)	690.2 (4.4)	682.3 (5.5)
4(16)	582.9	1573.8 (-170.0)	543.9 (6.7)	526.5 (9.7)
5(20)	515.9	1408.9 (-173.1)	469.0 (9.1)	455.3 (11.8)
6(24)	447.6	1266.4 (-182.9)	398.3 (11.0)	394.3 (11.9)
7(28)	415.7	1138.7 (-173.9)	367.2 (11.7)	**348.4 (16.2)**
8(32)	366.1	1037.4 (-183.4)	316.5 (13.5)	312.4 (14.7)
9(36)	306.1	953.8 (-211.6)	256.1 (16.3)	290.3 (5.2)
10(40)	287.5	903.9 (-214.4)	236.8 (17.6)	270.3 (6.0)
11(44)	259.7	855.1 (-229.3)	209.6 (19.3)	250.6 (3.5)
12(48)	248.6	865.7 (-248.2)	**197.2 (20.7)**	249.0 (-0.1)
13(52)	220.2	**784.5 (-256.3)**	181.4 (17.6)	221.0 (-0.4)
Scenario 2: TPC-DS Q52				
3(12)	719.9	2041.5 (-183.6)	660.8 (8.2)	716.0 (0.6)
4(16)	562.7	1763.2 (-213.3)	517.3 (8.1)	559.6 (0.6)
5(20)	471.8	1454.2 (-208.2)	412.7 (12.5)	468.3 (0.8)
6(24)	417.7	1335.4 (-219.7)	358.3 (14.2)	415.3 (0.6)
7(28)	364.1	1172.8 (-222.1)	304.7 (16.3)	**360.7 (0.9)**
8(32)	324.7	1068.4 (-229)	265.0 (18.4)	322.3 (0.7)
9(36)	306.8	1004.2 (-227.3)	247.0 (19.5)	**304.2 (0.9)**
10(40)	275.2	920.0 (-234.3)	215.2 (21.8)	273.1 (0.8)
11(44)	258.8	884.2 (-241.7)	200.2 (22.7)	257.0 (0.7)
12(48)	250.0	857.7 (-243.1)	**190.7 (23.7)**	248.3 (0.7)
13(52)	226.1	**805.6 (-256.3)**	179.3 (20.7)	224.2 (0.8)

of a constant to approximate server synchronization delays, which is applied to response times that already include delays caused by stage synchronization.

Both the Task Precedence model and dagSim simulator showed much better estimates, with errors ranging from 4.4% to 20.7% and from −0.1% to 16.2%, respectively. These models consider the parallel execution DAG to better capture the dependencies and interactions among stages, resulting in more accurate estimates of synchronization delays. For scenario 2, we found similar results when comparing the three models, as shown in Table 5.

Overall, taking absolute values, on average the errors were 201.14% for the Fork-Join model, 13.45% for the Task Precedence model, and 7.73% for dagSim in scenario 1. For scenario 2, the Fork-Join model obtained 225.33%, the Task Precedence model obtained 16.92%, and dagSim obtained 0.74%.

As described in Section 3, the Fork-Join model is a very simple approach, which depends only on the number of cores and on the average execution time of a single core for performance prediction. Unfortunately, as our results attest, its simplicity is not suitable for predicting performance, with reasonable accuracy, in the scenarios we are interested in. Similarly large errors were

Table 6 Real and predicted execution time quartiles, Q52, 500 GB

Cores	Quartile	dagSim [s]	Real [s]	Error [%]
12	Q_1	634.067	632.305	-0.28
12	Q_2	635.270	638.675	0.53
12	Q_3	636.564	644.211	1.19
24	Q_1	311.366	313.597	0.71
24	Q_2	312.357	315.881	1.12
24	Q_3	313.600	317.913	1.36
36	Q_1	224.383	225.795	0.63
36	Q_3	226.557	230.555	1.73
48	Q_1	165.070	165.319	0.15
48	Q_2	166.061	166.590	0.32
48	Q_3	167.221	168.646	0.84
52	Q_1	149.080	150.416	0.89
52	Q_2	149.980	151.188	0.80
52	Q_3	150.941	152.074	0.75

observed for all the other analyzed scenarios as well. The Task Precedence and dagSim models, on the contrary, provide very good estimates, given the complexity of the environment and workloads, especially for practical purposes of planning and managing the resource requirements. The somewhat larger errors of the analytical Task Precedence model are probably due to several sources of approximations embedded in the model (see Section 3.2.2 and [**?**]).

Therefore, for the following scenarios we report results only for the Task Precedence model and the dagSim simulator approaches. As these models explicitly capture the precedence relationships between stages, they may be able to better estimate the synchronization delays and provide more accurate predictions.

One advantage of dagSim over the analytical models (Task Precedence, in particular) is that it can provide estimates of not only average execution time, but also execution time percentiles, which can be quite useful for planning and managing resources based on probabilistic SLAs (e.g., the probability that the execution time exceeds a threshold τ is at most α). Table 6 reports, as an example, the quartiles of the execution times of Q52 on the 500 GB dataset, with D14v2 deployments. The table displays both the simulated quartiles and the ones derived from 50 sample runs on the real system. The estimated quartiles are quite accurate, with a worst case relative error of 1.73% and an average of 0.82%.

4.4 Results on the A3 VMs (Scenarios 3 & 4)

Scenarios 3 and 4 consider the same cluster sizes, dataset sizes and query workloads as scenarios 1 and 2. Yet, they differ from the latter on the VM type and on how the experiments were planned. The number of executors cores per nodes varied, A3 VMs have also less available RAM memory than the D12v2 ones. Thus the experiments in scenarios 3 and 4 simulate an environment with memory pressure. Across different runs with the same configuration a varying

number of executors were allocated. This was due to memory contention among the executors and the underlying operating system processes and the behavior was not deterministic.

Table 7 shows the results for both scenarios 3 and 4. Errors were computed a posteriori by considering the actual number of cores used during the execution. Regarding scenario 3, the prediction error of Task Precedence and dagSim varies from 0.8% to 10.0% and from 0.01% to −11.7%, respectively. Similar results were found also for scenario 4, as errors varied from 2.4% to 15.9% for Task Precedence, and from 0.3% to 11.6% for dagSim.

The results found in scenarios 3 and 4 support those observed scenarios 1 and 2, suggesting that the models are stable for both VMs and queries tested. Overall, taking absolute values, the errors were, on average, 5.03% for the Task Precedence model and 3.50% for dagSim in scenario 3. Corresponding values for scenario 4 are 9.8% and 1.17%, respectively. Thus, both models performed well under a deployment subject to memory pressure. The dagSim simulator proved stable, especially on scenario 4, with very small errors except for the 9 nodes and 30 cores experiment. The Task Precedence errors increased as the number of cores increased, indicating that the prediction can be affected by the cluster size. We assume that this is due to the accumulation of synchronization delay estimation errors present in the model.

4.5 Results on the D4v2 VMs (Scenarios 5, 6 & 7)

For scenarios 5, 6, and 7 we executed the Task Precedence model and dagSim simulator considering Spark 2.1.0 logs for a set of machine learning algorithms, namely K-Means, Logistic Regression, and SVM. The ML workloads are iterative algorithms and usually characterized by a larger number of stages than the queries in scenarios 1–4. For these applications, data partitions are cached and accessed multiple times during the iterations. As noticed, these workloads present a higher variability since each iteration consists of data processing and partitions recomputation in case of cache eviction.

As detailed in Table 8, for all three ML workloads, the prediction error of the Task Precedence model is inversely proportional to the size of the dataset, i.e., the larger the dataset, the smaller the prediction error. Since processing larger datasets requires more tasks to be executed, the experiments yield a lower variance on the application response times. Analogously, a smaller number of tasks would result in higher variance across multiple runs. Regarding the different cluster sizes, the results of Task Precedence are similar to those in the previous scenarios: the errors tend to increase with cluster size. As previously discussed, this is attributed to the accumulation of synchronization delays over a large number of distributed tasks running in multiple cores.

We further looked into the execution times measured for individual runs of each workload on each configuration and observed that the setup in which Task Precedence produced the largest errors for all three benchmarks (8 GB on 48 cores) coincides with the scenario with the highest variance across multiple

Table 7 Scenarios 3 & 4: Real and predicted execution times (seconds) for Q26 and Q52 on A3 VMs. Results in *bold* for the maximum error and *shaded cells* for the minimum error.

Nodes (cores)	Real	Task Precedence (error %)	DagSim (error %)
Scenario 3: TPC-DS Q26			
3(6)	2532.3	2512.8(0.8)	2538.5(-0.3)
3(8)	2071.2	2052(0.9)	2086.1(-0.7)
4(10)	1778.8	1763.6(0.9)	1778.6(0.01)
4(12)	1690.6	1674.5(1.0)	1704.5(-0.8)
5(14)	1439.3	1414(1.8)	1452.9(-0.9)
5(16)	1271.6	1243.3(2.2)	1281.0(-0.7)
6(18)	1127.2	1099.8(2.4)	1152.9(-5.4)
6(20)	1093.6	1064.2(2.7)	1095.2(-0.1)
7(22)	996.7	963.2(3.4)	1050.4(5.4)
7(24)	911.2	874.8(4.0)	933.9(-2.5)
8(26)	720.8	682.2(5.4)	735.6(-2.1)
9(30)	658.8	617.6(6.3)	691.6(-5.0)
9(32)	629.8	589(6.5)	643.7(-2.2)
10(34)	625.8	584.3(6.6)	647.6(-3.5)
10(36)	577.4	532.4(7.8)	602.9(-4.4)
11(38)	546.6	503.1(8.0)	572.1(-4.7)
11(40)	530.6	487.3(8.2)	555.2(-4.6)
12(42)	488.4	447.3(8.4)	510.6(-4.6)
12(44)	470.7	427.6(9.2)	493.8(-4.9)
13(46)	446.4	405.6(9.1)	455.2(-2.0)
13(48)	430.5	**387.5(10.0)**	**460.9(-7.1)**
Scenario 4: TPC-DS Q52			
3(6)	2158.9	2107.6(2.4)	2153.0(0.3)
3(8)	1709.2	1656.8(3.1)	1702.5(0.4)
4(10)	1327.4	1276.4(3.8)	1316.3(0.8)
4(12)	1124.5	1072.5(4.6)	1117.4(0.6)
5(14)	976.4	924.0(5.4)	970.5(0.6)
5(16)	884.9	832.6(5.9)	880.4(0.5)
6(18)	816.5	764.8(6.3)	809.8(0.8)
6(20)	738.8	687.3(7.0)	733.3(0.7)
7(22)	667.9	613.6(8.1)	663.3(0.7)
7(24)	620.1	566.2(8.7)	615.8(0.7)
8(26)	572.1	512.7(10.4)	568.4(0.6)
9(30)	525.9	466.6(11.3)	**465.0(11.6)**
9(32)	492.3	432.7(12.1)	487.9(0.9)
10(34)	462.2	402.5(12.9)	457.4(1.0)
10(36)	442.1	382.6(13.5)	439.3(0.6)
11(38)	438.7	380.2(13.3)	436.8(0.4)
11(40)	418.4	359.1(14.2)	415.6(0.7)
12(42)	392.1	334.2(14.8)	390.2(0.5)
12(44)	383.7	325.9(15.1)	379.5(1.1)
13(46)	378.4	318.8(15.8)	380.1(-0.4)
13(48)	362.8	**305.0(15.9)**	359.3(1.0)

Table 8 Scenarios 5, 6 & 7: Real and predicted execution times (seconds) for ML workloads on D4v2 VMs. Results in *bold* for the maximum error and *shaded cells* for the minimum error.

Nodes (cores)	Data set size (GB)	Real	Task Precedence (error %)	dagSim (error %)
Scenario 5: K-Means				
3 (24)	8	99.0	81.9 (17.3)	75.6 (23.6)
3 (24)	48	342.2	325.1 (5.0)	364.6 (-6.5)
3 (24)	96	862.1	845.9 (1.9)	788.4 (8.5)
6 (48)	8	90.3	**74 (18.1)**	70.3 (22.1)
6 (48)	48	195.0	178.8 (8.3)	219.2 (-12.4)
6 (48)	96	594.3	572.9 (3.6)	**746.2 (-25.6)**
Scenario 6: Logistic Regression				
3 (24)	8	164.6	159.5 (3.1)	156.1 (5.1)
3 (24)	48	669.4	664.4 (0.7)	671.7 (-0.3)
3 (24)	96	1418.8	1414.1 (0.3)	1404.9 (0.9)
6 (48)	8	166.5	**161.0 (3.3)**	**156.5 (6.0)**
6 (48)	48	368.2	362.5 (1.5)	362.9 (1.4)
6 (48)	96	1200.7	1192.6 (0.6)	1193.9 (0.5)
Scenario 7: SVM				
3 (24)	8	190.9	171.7 (10.1)	167.7 (12.2)
3 (24)	48	356.7	339.2 (4.9)	358.1 (0.4)
3 (24)	96	1,367.0	1349.6 (1.3)	1323.9 (3.2)
6 (48)	8	189.7	**170.2 (10.3)**	**164 (13.5)**
6 (48)	48	372.5	353.2 (5.2)	352.2 (5.4)
6 (48)	96	650.5	631.1 (3.0)	635.4 (2.3)

runs. The large number of cores used on a relatively small dataset, which might occasionally cause resource underutilization, may explain the slightly worse performance of the model in this setup. In contrast, we see no clear error pattern for dagSim.

With respect to absolute errors, both Task Precedence and dagSim provide very good prediction accuracy across the considered set of experiments, covering different platforms and configurations. Specifically, taking absolute values, the Task Precedence model achieved an average error of 9.03%, 1.62%, and 5.80% for scenarios 5, 6 and 7, respectively, with an overall average of only 5.48%. DagSim, in turn, produced errors equal to 16.45%, 2.42%, and 7.42% for scenarios 5, 6, and 7, respectively, with an overall average of 8.76%.

4.6 Summary of Results

To summarize the accuracy results of the two most promising approaches — Task Precedence and dagSim — we observe that the Task Precedence model achieved errors that vary from 0.8% to 20.7%, being on average only 7.38% (average computed across all errors taken in absolute values). The errors achieved

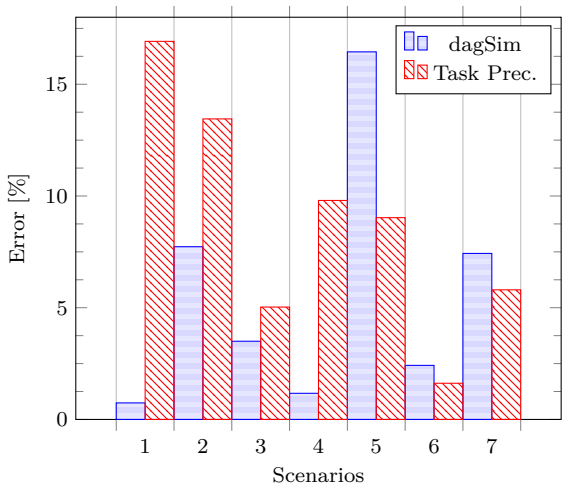

Figure 6 Average prediction errors across all analyzed scenarios (averages computed across errors taken in absolute values)

by dagSim, on the other hand, vary from 0.7% up to −25.6%, but with an average of only 5.65% (see Figure 6). It is important to observe that in the performance evaluation literature, 30% errors (consistent across cluster size) in execution time predictions can be usually expected, especially from analytical models (see [**?**]).

Thus, both approaches can be considered suitable for predicting the performance of big data applications. Moreover, we noticed that dagSim outperforms the Task Precedence model in all scenarios with interactive queries, whereas the latter was the best approach for the iterative ML algorithms.

Our results of the Fork-Join model, on the other hand, indicate that the simpler model provides only a very coarse approximation, which is too conservative, especially compared to the other approaches.

Regarding execution times, we note that both models ran very quickly and are suitable for online predictions. The average execution time of dagSim ranges between 0.76 s and 3.26 s for scenario 1 to 4, with very low variability across multiple runs, with a coefficient of variation[9] (CV) of lower than 6%. Vice versa, JMT took on average from 25 to 115 seconds.

For scenarios 5, 6 and 7, despite the higher variability (CV ranges between 0.85 and 0.9), the average execution times for dagSim were still short, in 1.2-4.9 seconds range. Note that in this latter scenarios the higher variability was due to the different size of the underlying dataset (which has an impact on the number of tasks within stages and the number of simulated events).

The execution times of the analytical Task Precedence model was very short, varying from only 4.18 ms (for scenario 4) to up to 85.71 ms (for scenario 7). They were also mostly stable (i.e., low CVs) across all scenarios.

[9] Ratio of standard deviation to mean value.

Table 9 Average execution time and CVs for each scenario

Scenario	Task Precedence		DagSim	
	Avg. [ms]	CV [%]	Avg. [ms]	CV [%]
1	5.35	11.96	3,085.29	5.54
2	4.59	4.81	763.45	5.97
3	6.26	3.95	3,264.96	1.43
4	5.08	2.72	815.47	2.70
5	9.42	31.97	1,238.11	91.56
6	28.38	29.20	2,417.94	84.10
7	59.82	36.64	4,923.00	87.35

Thus, comparing both tools, dagSim's execution times exceed those of the analytical model by some orders of magnitude: their ratio varies from around 80 to over 580, about 250 on average. However, the Task Precedence model is limited to assess average execution time, whereas dagSim can provide also percentiles of application performance, thus enabling much finer-grained analyses.

5 Conclusions and Future Work

In this paper, we compared two analytical models and proposed an ad hoc simulator for the performance prediction of Spark applications running on cloud clusters.

We evaluated all three models in multiple cloud configurations and workloads, including SQL and iterative ML benchmarks. Our results indicate that the Fork-Join model is too inaccurate to be considered in practice. On the other hand, both the Task Precedence model and the dagSim simulator perform very well for predicting the average application execution time, both in terms of prediction accuracy and model execution time. Thus, both are quite effective in capturing the dynamic resource assignment implemented in Spark, although dagSim seems somewhat better for interactive queries, whereas Task Precedence outperforms it for the machine learning workloads.

In our future work, we plan to extend our models to cope with scenarios where multiple applications run concurrently competing to access the resources in the same clusters. We also intend to embed the models into a run time optimization tool for managing dynamically cloud resources with the aim of providing application execution within an a priori fixed deadline while minimizing cloud operational costs.

Acknowledgement

The authors' work has been partially funded by the EUBra-BIGSEA project by the European Commission under the Cooperation Programme (MCTI/RNP

3rd Coordinated Call), Horizon 2020 grant agreement 690116. This research was also be partially funded by CAPES, CNPq and FAPEMIG, Brazil.

Spark experiments have been supported by Microsoft under the Top Compsci University Azure Adoption program.