Quick Execution Time Predictions for Spark Applications

Sarah Shah*, Yasaman Amannejad†, Diwakar Krishnamurthy*, Mea Wang§
*Department of Electrical and Computer Engineering, University of Calgary, Calgary, Canada
†Department of Mathematics and Computing, Mount Royal University, Calgary, Canada
§Department of Computer Science, University of Calgary, Calgary, Canada
yamannejad@mtroyal.ca, {sarah.shah1, dkrishna, meawang}@ucalgary.ca

***Abstract*—The Apache Spark cluster computing platform is being increasingly used to develop big data analytics applications. There are many scenarios that require quick estimates of the execution time of any given Spark application. For example, users and operators of a Spark cluster often require quick insights on how the execution time of an application is likely to be impacted by the resources allocated to the application, e.g., the number of Spark executor cores assigned, and the size of the data to be processed. Job schedulers can benefit from fast estimates at runtime that would allow them to quickly configure a Spark application for a desired execution time using the least amount of resources. While others have developed models to predict the execution time of Spark applications, such models typically require extensive prior executions of applications under various resource allocation settings and data sizes. Consequently, these techniques are not suited for situations where quick predictions are required and very little cluster resources are available for the experimentation needed to build a model. This paper proposes an alternative approach called PERIDOT that addresses this limitation. The approach involves executing a given application under a fixed resource allocation setting with two different-sized, small subsets of its input data. It analyzes logs from these two executions to estimate the dependencies between internal stages in the application. Information on these dependencies combined with knowledge of Spark's data partitioning mechanisms is used to derive an analytic model that can predict execution times for other resource allocation settings and input data sizes. We show that deriving a model using just these two reference executions allows PERIDOT to accurately predict the performance of a variety of Spark applications spanning text analytics, linear algebra, machine learning and Spark SQL. In contrast, we show that a state-of-the-art machine learning based execution time prediction algorithm performs poorly when presented with such limited training data.**

***Keywords*— Apache Spark, Big Data Processing, Performance Prediction, Performance Engineering, Scalability**

I. Introduction

The Apache Spark cluster computing platform [1] is being increasingly used to develop big data analytics applications. Execution time of Spark applications is an important concern for users and operators of a Spark cluster. Cluster users are typically interested in ensuring that their application meets a desired execution time target. Cluster operators, on the other hand, would like to provision just enough resources to applications such that application execution time targets are satisfied while simultaneously maximizing utilization of cluster resources and reducing costs. Spark cluster users and operators can benefit from an execution time prediction tool that would provide quick insights on how the execution time of any given application is likely to change as a function of the resources allocated to the application and the size of data that needs to be processed. While Spark provides sophisticated tools to monitor application performance [2], it does not yet support such a prediction tool.

There have been a number of recent efforts at modeling and optimizing the performance of Spark applications using statistical and machine learning techniques [3]–[8]. To be effective, these techniques require extensive performance data from previous executions of an application covering a range of different resource allocations and data sizes of interest to build models. In the absence of such historical data, controlled experiments are needed for building the models. Conducting such experiments is time consuming and resource intensive. Consequently, the use of these existing modeling techniques might not be feasible in situations where quick predictions are desired and extensive access to cluster resources for the experimentation needed to build the models is impractical. In this paper, we aim to reduce the time and resource requirements of the prediction process to address such situations.

We propose a lightweight, analytic execution time prediction approach called PERIDOT, **PER**formance pred**I**ction mo**D**el f**O**r Spark applica**T**ions. PERIDOT executes an application under a fixed resource allocation with two different-sized, small subsets of its input data. Based on observations on the internal dependencies in the application as well as the impact of data partitioning and data size on execution times in these runs, PERIDOT deduces the execution times of the application under other resource allocation settings and input data sizes. By relying on just two runs per application and using small datasets, PERIDOT significantly reduces the time and resources required to construct application models thereby facilitating quick insights into the performance behaviour of applications.

We evaluate PERIDOT using 8 well-known big data applications spanning text analytics, linear algebra, machine learning and Spark SQL executing on a Spark cluster. Our results indicate, as expected, that naive prediction techniques that assume ideal application speedup yield poor accuracy. In contrast, models derived using PERIDOT yield very good accuracy for all 8 applications over a range of resource allocations and input data sizes. Specifically, PERIDOT yields a mean prediction error of 6.6% over all applications. In particular, results show that PERIDOT is able to accurately capture the impact of complex internal application dependencies such as those observed in many Spark SQL queries. Generating models and predictions using PERIDOT is quick. For instance, while exhaustive experimentation of all possible configurations we explore for our applications requires over $2,500$ core hours, PERIDOT requires only

978-3-903176-24-9 © 2019 IFIP

0.7% of this effort.

We also compare PERIDOT with a state-of-the-art machine learning based prediction approach [7] that is agnostic to an application's internal data dependencies and partitioning mechanisms. Specifically, we compare the ability of this technique and PERIDOT to offer accurate predictions with limited experiment data. Our results show that in contrast to PERIDOT, the machine learning technique yields poor predictions when only limited training data is available. These results suggest that PERIDOT is feasible and effective in offering quick, resource-efficient, and accurate predictions. It can be used by Spark users and embedded within job schedulers to quickly configure any given Spark application at runtime for a desired execution time. We have made available all the execution scripts and logs of the experiments in this paper online [9].

II. Apache Spark Platform

A Spark *application* consists of two types of operations, namely *transformations* and *actions*. A transformation applies a function on each element of a distributed collection of objects called a *Resilient Distributed Dataset (RDD)*. A transformation can cause one or more additional RDDs to be created. Actions trigger the execution of functions associated with one or more transformations to produce meaningful results. For each action in an application, a Spark *job* is created and executed. A single application can trigger multiple jobs which run in sequence. If an application has multiple threads, jobs can execute in parallel.

A Spark job can be separated into one or more physical units of execution called *stages*. A Spark stage may depend on the output data generated by previous stages or might be independent of other stages. The Spark runtime will schedule a stage for execution only after all stages that stage is dependent on have finished executing. In contrast, stages that have no dependencies with one another can be scheduled to run concurrently thereby resulting in *parallel stages*.

Dependencies among application stages are represented within Spark as a *Directed Acyclic Graph (DAG)*. A DAG contains a set of vertices and edges, where any given vertex represents a RDD and its edges represent operations to be applied on the RDD. When an application starts, the application's DAG is created and submitted to Spark's DAG Scheduler for execution. Fig. 1 provides our simplified visualization of the DAG for Query52, a Spark SQL implementation of a query in the TPC-DS [10] benchmark suite. In this figure, rectangles represent jobs and circles represent application stages. From this excerpt, one can observe multiple stages within a job. Some stages run in parallel while others run in sequence. Regardless of the size of input data or the resource allocation setting, the overall DAG structure remains the same for any given application. As described in Sec. IV, PERIDOT estimates the DAG-induced dependencies from an application execution that uses a small subset of its input data. It leverages this information while estimating execution times for other data sizes and resource allocation settings.

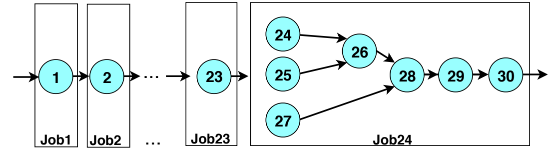
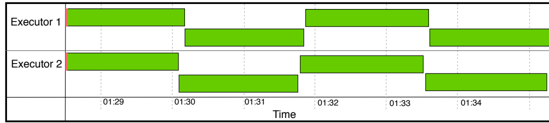


Fig. 1. DAG structure for Query 52

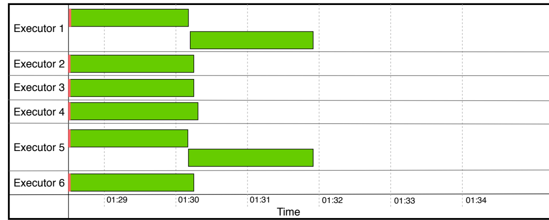
We next focus on behaviour within each stage. Within each stage, to facilitate parallel processing, Spark splits the input data of each stage into smaller data partitions that are typically of equal size. By default, the number of partitions is $\lceil \frac{D}{block} \rceil$, where D is the input size and $block$ is the block size employed by the underlying file system. However, as observed in many of the TPC-DS applications in our study, a stage can override this default partitioning behaviour by explicitly specifying the number of partitions. A stage consists of a set of *tasks* with each task applying the operations of the stage on one partition of the data. A task is executed by one of the executors allocated by Spark to that stage. An *executor* is a process that runs on a *worker node* in the cluster, i.e., a node that executes application code. A worker node can host one or more executors. Each executor can be allocated a configurable amount of worker node resources such as processing cores and memory.

Assuming only one stage is scheduled for execution, the number of concurrent tasks of that stage executing at any given time depends on the number of executor cores assigned to the stage, and the number of partitions to be processed in the stage. We refer to the pattern of execution of tasks within a stage as a task *wave*. We define the wave time as the time taken for one wave of execution. Fig. 2 derived from Spark's visualization of actual stage executions shows two examples of tasks running in waves. In Fig. 2a, two executors with one core each are assigned a total of eight tasks. The tasks are equally divided among the two executors, resulting in four *full waves*, i.e., waves that fully utilize the available parallelism. In Fig. 2b, the same eight tasks are distributed among six executors. This increase in the number of executors reduces the number of waves to two. However, there are four idle executors during the second wave. We refer to this wave as a *partial wave* because it does not fully utilize the parallelism available. When multiple stages are scheduled for execution concurrently, tasks from different stages may execute at the same time as part of a wave. Consequently, the individual task execution times in a wave are more heterogeneous than in the single stage scenario.

For a given number of executors, increasing the input data size of a stage typically increases the number of partitions. Depending on the exact number of executors, this can either cause the number of waves to remain the same or increase. The number of waves in a stage increases by one or more if the number of additional partitions exceeds the number of idle cores in the partial wave of that stage. In contrast, the number of waves remains the same if the number of additional partitions is less than or equal to the number of idle cores in the partial wave. In this case, the increase in data size has the impact of "filling up" the partial waves,



(a) Full wave



(b) Partial wave

Fig. 2. Examples of task waves

thereby utilizing the available parallelism more effectively.

From the above discussion, the number of waves in a stage and hence the stage execution time depends simultaneously on the number of executors and data size. This suggests that the impact of both the number of executors and size can be modeled by estimating the number of waves and wave times in a stage. We exploit such an approach within PERIDOT.

III. Related work

There have been a number of recent efforts on modeling and optimizing the performance of Spark applications [3]–[8], [11]–[14]. Singhal and Singh [15] developed a simulation model that captures in detail Spark's internal job processing mechanisms. The model's parameters are derived from executions of a given application under various configuration settings. The authors showed how the parameterized model can then be used to simulate the application's scaling behaviour. Petridis et al. [3] focused on 12 Spark configuration parameters and assessed their impact using measurements. Based on these measurements, the authors proposed a trial-and-error performance tuning methodology for Spark applications. Wang et al. [4] proposed a machine learning based parameter tuning approach using multi-class classification. For each application, they generate a list of 500 execution records where each record contains a list of Spark configuration settings and the execution time achieved with those settings. These records are used to train a model that can predict execution times under arbitrarily specified configuration settings. These techniques consider a large parameter space and hence require a large amount of prior execution data to be really effective. Unfortunately, obtaining such data might not always be feasible. For example, Jyothi et al. [16] analyzed workloads on production clusters and showed that about 40% of applications were non-recurrent, i.e., did not have historical execution data.

Similar to this paper, some studies focused exclusively on the impact of data size and the amount of resources assigned to a Spark application. Gibilisco et al. [5] executed a given Spark application with different, smaller subsets of the input data. They trained multiple polynomial regression models from these executions and selected the model with the least error. Finally, the model obtained in this manner is used to predict the execution time of the application when it processes the entire input data. Wang and Khan [11] collected detailed performance logs, e.g., execution time, memory consumption, I/O overheads, for the stages in a Spark application. They exploit these logs to simulate how the application's execution time will change with input data size. Gounaris et al. [6] proposed regression models that capture how execution time changes with the number of nodes allocated to the application. In contrast to these approaches, PERIDOT can simultaneously capture the impact of *both* data size and number of executor cores.

Venkataraman et al. [7] proposed an approach called Ernest that is designed to simultaneously predict the impact of input data size and the number of worker nodes. Ernest uses an optimal experiment design method [17] to select executions with different data sizes and numbers of nodes. These executions are used to train a regression model for the application that has data size and number of nodes as independent variables. Islam et al. [18] proposed an approach called dSpark for modeling execution time with respect to the number of executors and input data size. Similar to Ernest, dSpark uses a set of controlled executions to train a regression model. Both these techniques report very good accuracies when there is extensive experiment data to train the regression models. Other machine learning based approaches have reported similar results [12], [13]. We note that in contrast to PERIDOT both these techniques do not exploit knowledge of spark internals such as the internal stage dependencies of applications, Spark's partitioning mechanisms, and the impact of task waves. We show in this paper that considering such information allows us to extract models that can provide accurate predictions with very little experimentation effort.

Amannejad et al. [14] outlined an execution time prediction approach that combined Amdahl's law [19] with knowledge of Spark's partitioning behaviour. Their approach required 2 reference executions at the same size but different executor settings. In contrast to this work, they considered only a limited set of applications, which did not exhibit the complex dependencies and parallel stages that we observe in this work. Furthermore, unlike this work, they validate their predictions only on a single-node setup and do not consider cluster environments.

In conclusion, most existing studies require extensive experimental data to build execution time models. We explore in this work an alternative technique that can offer quick predictions based on limited experimental data.

IV. PERIDOT

PERIDOT does not require exhaustive prior executions of an application to derive a model. Given an application and its input data, PERIDOT launches just two different executions of the application using smaller subsets of the input data so as to not consume a significant amount of cluster resources. The two executions measure the behaviour of the application under two different input data sizes and the same number of executor cores. Using these two executions

we follow a three step process to predict the execution time of the application with different input sizes and executor cores. We first estimate the dependencies among stages in the application. Using the execution logs, we also characterize the number of partitions that each stage processes and whether the number of partitions changes with size (Sec. IV-A). Second, we estimate the number of waves that each stage or a group of parallel stages processes and use that to calculate their wave times (Sec. IV-B). Finally, wave times and other parameters extracted from these execution logs are used for predicting the execution time of the application with different input sizes and executor core numbers (Sec. IV-C). We next describe the details of each step.

A. Application Dependency Identification

In this step, we run the Spark application twice with two small input data sizes and the same number of executor cores. We call these two runs as *reference executions*. We collect the execution logs of both reference executions. The execution log records detail information such as stage id, tasks in the stage, the time at which a task is scheduled, the time at which the task obtains resources and starts executing, and the time at which the task finishes. From the reference logs, we estimate the dependencies between the stages and characterize how the application's partitioning behaviour changes with size. As mentioned earlier, the dependency structure depends on the operations defined in the application and remains the same even when the input data or the executor cores assigned to the application are changed.

We next identify groups of stages and represent the application's dependency as a sequence of these groups. We define a *stage group* as all stages scheduled at the same time. A stage group may contain only one stage, e.g., a sequential stage, or multiple stages, e.g. parallel stages. In Fig. 1, as an example, no other stage is scheduled to run in parallel with stage 1 and therefore we create a stage group with only one stage. Stages 24, 25, and 27 are parallel stages since they are scheduled in parallel and hence we create one stage group with these stages in it.

Since parallel stages are scheduled at the same time, they compete for the processing resources assigned to the application. Consequently, the waves contain tasks from different stages and hence are more complex than stage groups that contain only a single stage. Regardless of whether a stage group is sequential or parallel, we define the execution time S_i of the stage group i as the difference between the time the last task in the group completes and the time the first task in the group is scheduled. Apart from capturing the computation time of tasks, this metric also includes delays due to tasks waiting for executor cores. The process of identifying stage groups based on stage scheduling time yields a sequence of stage groups estimating the dependency structure of the application.

We note that this method of extracting application dependency based on stage scheduling times is a simplification. In particular, there might be scenarios where a stage group might be able to execute even before the previous stage group has finished executing. Considering Fig. 1, Spark's DAG scheduler will schedule stages 24, 25, and 27 at the same time. Accordingly, our approach will place these stages within a single group followed by stage 26 in the following group. However, from the DAG, tasks scheduled from stage 26 can start executing even before tasks from stage 27 complete if there are idle cores available. An exact recreation and capturing of the dependencies would require a detailed simulation of the DAG's scheduling and execution. In contrast, our simplified approach allows us to analytically estimate execution times without resorting to simulation. The ability to offer quick predictions makes our approach well-suited for use within job schedulers at runtime for "right sizing" an application. Furthermore, the analytic nature of our approach allows it to be used within job planning optimization frameworks. We evaluate the effectiveness of our simplification in Sec. VI.

After extracting the sequence of stage groups from each reference execution log, we compare the corresponding groups in the references to characterize partitioning behaviour. Specifically, we label each stage group as either a *fixed partition* or a *variable partition* stage group. The number of partitions processed by a fixed partition stage group does not change when the input data to the application changes. From our empirical observations, the execution time of such stage groups typically does not vary when the input data size changes. Consequently, the execution times of such a stage group can be modeled as a constant. In contrast, the number of partitions processed by a variable partition stage group changes with respect to the input data size. Its execution time hence changes with data size. Since the two reference executions use different input data sizes, it is possible to identify the fixed and variable stage groups.

To summarize, the inputs of this step are the execution logs of the two reference executions and the outputs are the sequence of stage groups labeled as fixed or variable partition groups. Moreover, we extract the number of partitions in each stage group under each of the two reference executions. We also extract the stage group execution times and the end to end execution times from both references.

B. Model Parameter Extraction

Based on the outputs from the previous step, we first calculate the mean wave time parameter for each variable stage group. As we show in Sec. VI, the wave time of any given stage in an application remains almost the same regardless of the size of the input data and the number of cores allocated to the stage provided the structural properties of the data, e.g., number of features in a machine learning dataset, remain unchanged. Consequently, knowledge of the mean wave time would allow us to offer predictions for any executor cores and data size as we outline later in Sec. IV-C.

The number of waves N_i in stage group i can be estimated using (1) where P_i is the number of partitions in the stage group and E is the number of executor cores assigned to the application. The wave time W_i can then be calculated by dividing the total execution time of the stage S_i by the number of waves N_i, as shown in (2). We calculate W_i in this manner using each of the two references. We then use

the mean of these two values $\hat{W}_i$ as the mean wave time parameter for stage group i.

$$N_i = \lceil \frac{P_i}{E} \rceil \quad (1)$$

$$W_i = \frac{S_i}{N_i} \quad (2)$$

We also record as parameters the mean number of partitions $\hat{P}_i^r$ in each stage group of the reference executions r. This is calculated as the sum of the corresponding P_i^r values in both references divided by the mean of the input data sizes of the two references $\hat{D}^r$. As shown later in (3), we use $\hat{D}^r$ to scale the number of partitions in each stage group while predicting for other input data sizes.

Finally, we estimate the parameter F_i representing the aggregate execution time contribution of the fixed partition stage groups. Consider an application execution with end to end execution time T. We sum the variable stage group execution times, i.e., the S_i values, and subtract this sum from T to obtain F_i. This process is carried out on both reference executions to obtain a mean fixed execution time $\hat{F}_i$.

C. Predicting Execution Time

Assume that an execution time prediction is desired for an input size of D^{new} under an executor core setting of E^{new}. For each variable stage group i, we estimate the new number of partitions that result under these settings using (3). This equation assumes that the number of partitions scales linearly with data size, which is the behaviour of Spark's default partitioning algorithm.

$$P_i^{new} = \frac{D^{new}}{\hat{D}^r} \times \hat{P}_i^r \quad (3)$$

We next estimate the number of waves N_i^{new} using (1). We use (2) to estimate the execution time S_i^{new} of the stage group using the mean wave time parameter $\hat{W}_i$ estimated previously. This process is repeated for all variable stage groups, var, and the sum of the S_i^{new} values is added to the fixed partition execution time parameter $\hat{F}_i$ estimated previously to obtain the execution time prediction T^{new}, as shown in (4).

$$T^{new} = \sum_{i \in var} \lceil \frac{P_i^{new}}{E^{new}} \rceil \times \hat{W}_i + \hat{F}_i \quad (4)$$

D. Discussion

PERIDOT is compact since it captures both executor and data scaling using the number of waves as shown by (1). Modeling in terms of the number of waves and mean time per wave allows PERIDOT to rely on just two reference executions. Moreover, the two reference executions are enough to identify the dependency structure of an application and incorporate its impact on stage group wave times and hence the overall execution time. To facilitate quick predictions, PERIDOT makes several simplifications with regards to the dependency structure, the characterization of fixed and variable partition stages, and the wave behaviour of stage groups. We next experimentally study the impact of such simplifications on accuracy.

V. Experiment setup

A. Experiment Testbed

We use PERIDOT to predict the execution time of applications in a large cluster setting. Specifically, we use a multi-node testbed consisting of nodes from the ARC cluster of University of Calgary [20]. Each node used from this cluster has a four socket AMD Istanbul processor. A socket contains 6 cores each clocked at 2.4 GHz. The 24 cores associated with one compute node share 250 GB of RAM. This environment is equipped with Spark 2.2.0 and operates based on a common network file system.

B. Experiment Factors

Applications - We evaluate our technique with 8 standard benchmarks encompassing text analytics, linear algebra, machine learning and Spark SQL. Word Count (Wc) counts the number of occurrences of each word in a text file. The input consists of random words generated from words in the Linux dictionary. Wc is a representative of a compute-intensive application. Sort (So) ranks records by their keys. The input is a set of samples, each represented as a numerical vector. Sort is both compute and memory intensive. Linear Regression (Lr) models the relationship between a dependent variable and a set of independent variables. Lr is a representative of a machine learning application that triggers multiple jobs. Correlation analysis (Co) calculates the statistical dependency of input variables. While Wc, So, Lr and Co are part of the standard Spark distribution, to expand our evaluation, we have also used a Matrix Multiplication (Mm) application [21]. Mm receives a $m \times n$ ($m >> n$) matrix, and multiplies the transpose of this matrix with itself to output a $n \times n$ matrix.

To further diversify our applications, we also use Spark SQL queries from the industry-standard TPC-DS benchmark suite [10]. We include these queries because of their more complex DAG structures compared to the rest of the applications. Specifically, we use Query26 ($Q26$), Query52 ($Q52$) and Query64 ($Q64$). $Q26$ and $Q52$ are interactive queries that have been used by other researchers developing performance models [22]. $Q64$ represents a particular stress case for PERIDOT. It performs complex joins on four different tables to trace patterns in data with respect to selected features. To the best of our knowledge, existing prediction approaches have not used $Q64$ in their validation exercises.

Table I shows some high-level characteristics for the selected applications. These applications vary in terms of the number of jobs, number of stages, DAG structures and the operations used. From the table, the applications we have selected cover the three most frequently observed communication patterns in Spark [7] namely, Collect, Tree Aggregation and Shuffle.

Input and resource configurations - We vary the input size from 1 GB to 100 GB. To process this data, we employ up to 40 executor cores distributed over 1 to 8 machines,

Table I. APPLICATION CHARACTERISTICS

	# of Jobs	# of Stages	Parallel Stages?	Collect	Tree Aggregate	Shuffle
Wc	1	2	No	✓	–	✓
Mm	2	2	No	–	–	✓
So	2	2	No	–	–	✓
Co	33	53	No	–	✓	✓
Lr	7	9	No	–	✓	✓
$Q26$	25	35	Yes	✓	✓	✓
$Q52$	25	31	Yes	✓	✓	✓
$Q64$	26	75	Yes	✓	✓	✓

with each machine running an executor with 5 cores. Each executor is configured with 50 GB of RAM. The choice of input sizes and executor core numbers allows us to fully exploit the in-memory data processing offered by Spark with no disk spills while ensuring sufficient spare memory is available for the Spark driver, i.e., the JVM process that coordinates the executors, and the operating system.

Other approaches - We first compare PERIDOT with an Ideal Scaling (IS) baseline. The IS baseline assumes that the performance ideally scales with executor cores. For example, for any given input size, doubling the executor cores assigned to an application results in half the execution time. Similarly, IS assumes that for a given number of executor cores the execution time increases linearly (with a unit slope) with input size.

We also compare PERIDOT with an existing machine learning based prediction approach from literature namely, Ernest [7]. As shown in (5), Ernest uses a regression model where an application's execution time is expressed as a linear combination of four components namely, a constant term, a linear dependence on the ratio of input size D to the number of nodes $nodes$ allocated to the application, a non-linear dependence on $nodes$ that captures inter-node communication overheads, and a linear dependence on $nodes$.

$$T = \theta_0 + \theta_1 \times (D \times \frac{1}{nodes}) + \theta_2 \times log(nodes) + \theta_3 \times nodes \quad (5)$$

While Ernest is shown to be effective in settings with extensive historical, i.e., training, data, we specifically focus on its ability to offer predictions from limited reference executions. We note that in contrast to PERIDOT, IS and Ernest do not explicitly consider application dependencies and the effect of task waves in their models.

C. Experiment Process

For each experiment, we have an initial phase where the two reference executions are carried out as described in Section IV to extract the application dependencies and the model parameters. Our reference executions use 5 cores. We use 1 GB and 2 GB data sizes in the reference experiments. In the evaluation phase, we run each Spark application with all executor core settings and input sizes to collect the application execution times. We repeat each measurement multiple times and use the mean of the measured execution times. We compare the measured execution times with those predicted by PERIDOT and the other techniques and report the relative error.

VI. RESULTS

This section is organized as follows. We first provide an overview of the performance of the 8 applications in Section VI-A. Section VI-B evaluates PERIDOT and compares its performance with the other two prediction techniques. We have made the execution scripts and the logs of all experiments available online [9].

A. Application Performance Characteristics

We first show how the execution times of the 8 applications change with the number of executor cores. On our setup, the execution times vary by factors of 9 to 24 while using the executor cores and size settings presented in Sec. V-B. As shown in Fig. 3[1], for all applications, execution time speedups become progressively less dramatic as the number of cores increases. This motivates the need for models that capture such trends so that the appropriate number of executors can be selected for any given application.

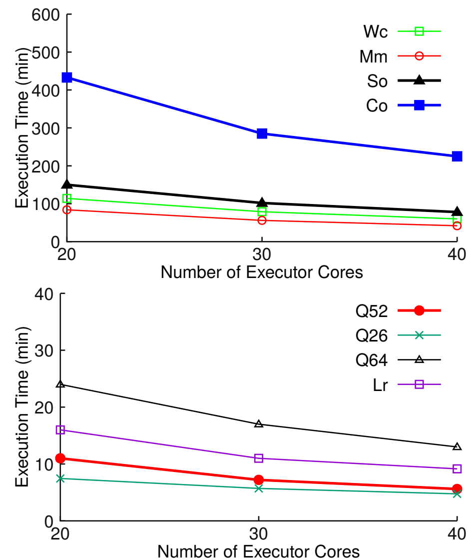


Fig. 3. Execution time of applications

The mean wave times can vary significantly across the various stage groups in an application. Fig. 4 shows this behaviour for Wc, Mm, and So in the reference executions. Other applications show similar behaviour but we omit the figures since they have a larger number of stages. This motivates PERIDOT's choice of modeling individual stage groups and aggregating the results of the individual stage group models.

Next, we focus on the wave behaviour of a given stage group across different executions. PERIDOT assumes that the prediction for a target execution can be obtained from the wave time of stage groups in the reference execution. The wave times of the reference and target are similar due to the following two reasons:

First, the block size remains almost the same in any given system. Hence, the system processes same sized blocks across different settings with only the number of blocks, i.e.,

[1]Due to large differences in application execution times, we have shown the results in two different graphs for better visibility

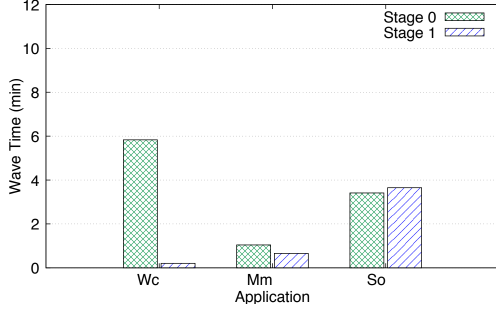


Fig. 4. Wave time of the different stages of the applications

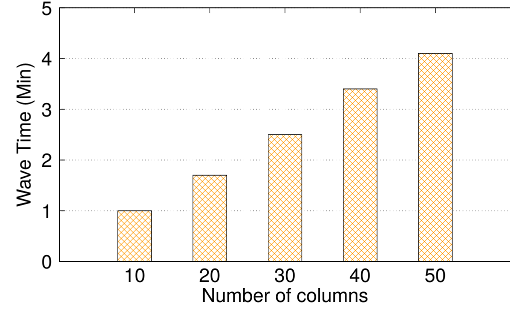


Fig. 6. Effect of number of columns on wave time

partitions, processed changing depending on the size of the application's input data. Since a wave is made of equal-sized, concurrent blocks, the wave times are likely to be similar across executions (provided the structure of the data is preserved, as discussed later). Fig. 5 plots the wave times of the largest stage of Mm and So under different executor core and size settings. This figure confirms that the wave time of any given stage group is very similar across different settings thereby justifying our modeling choice. We observe similar behaviour in other applications including those with parallel stages.

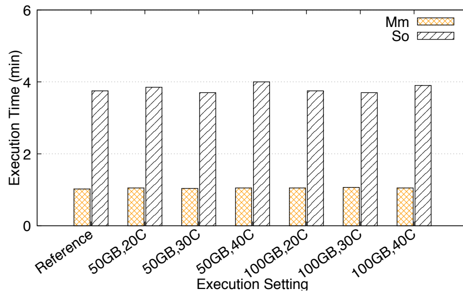


Fig. 5. Wave time of the applications with different settings

Second, the reference execution uses data sampled from the target data and the sampled data has the same structural characteristics as the original data. We note that the wave times of the reference execution may not provide a reliable estimate of the wave times of the target execution if the structural characteristics of the data across these 2 executions differ. For example, Fig.6 shows that the average wave times of the longest stage of Mm increases as the number of columns in the input matrix increases even though the block sizes are the same for all executions. Consequently, the data used in the reference executions should have the same number of columns as the data of the target execution for which a prediction is desired.

Finally, Table II shows PERIDOT's classification of stage groups for the different applications. From the table, PERIDOT identifies parallel stages for the Spark SQL queries. All of these are also tagged as variable stage groups. The number of stages encompassed in these stage groups is 5, 3, and 15 for $Q26$, $Q52$ and $Q64$, respectively. PERIDOT does not identify parallel stages in the other applications but tags several variable stage groups. From the table, many stage groups are tagged as fixed in all applications. Almost all of these are stage groups of very short execution times involving Spark operations such as *collect*, *first*, *take* and *runJob*. The overall execution times of all applications are dominated by the execution times of variable stage groups. For example, the variable stage groups contribute on average 68%, 83% and 67% to the overall execution times of $Q26$, $Q52$ and $Q64$, respectively. Although the contribution of fixed stage groups to the execution time is smaller, considering them in the prediction process improves accuracy.

Table II. STAGE TYPES CHARACTERIZATION

	Parallel Stages	Variable Stage Groups	Fixed Stage Groups
Wc	0	2	0
Mm	0	1	1
So	0	2	0
Co	0	22	31
Lr	0	6	3
Q26	1	1	30
Q52	1	1	28
Q64	1	1	60

B. *Accuracy of PERIDOT*

Fig. 7 shows the mean prediction errors of PERIDOT for the 8 applications. We perform predictions for all executor and data size settings that are not used by the reference executions. The mean prediction error for all applications are below 15%. The highest prediction error is for $Q52$ where the error goes up to 20% with the mean prediction error for this application being 13%. The accuracy of PERIDOT for applications with and without parallel stages is 10.4% and 6.0%, respectively. Considering all applications, the mean error is 6.6%. We did not notice any clear correlation between the magnitude of errors and the data size and executor core settings. PERIDOT is effective across a diverse range of settings. These results validate the modeling choices described in Section IV.

PERIDOT provides quick predictions. For instance, exhaustive experimentation of all possible configurations we explore for all 8 applications requires over $2,500$ core hours. The core hours needed by PERIDOT are only 0.7% of that incurred by exhaustive experimentation.

Fig. 8 compares PERIDOT and IS. PERIDOT significantly outperforms IS. For the Wc, Mm, So and Co applications, the mean accuracy of PERIDOT is 1.35 times that of IS. IS performs extremely poorly with Lr and the Spark SQL applications, which have complex DAGs. For $Q64$, which

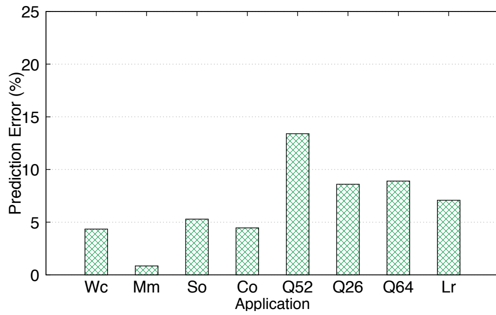


Fig. 7. Prediction errors of PERIDOT

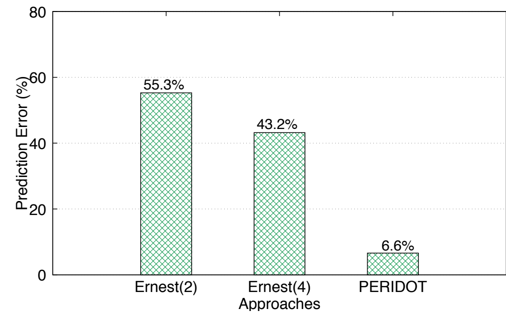


Fig. 9. Average prediction errors of PERIDOT and Ernest

has the most complex DAG, the prediction error of IS reaches 104%. PERIDOT's mean accuracy is over 10 times that of IS for these applications.

Finally, we compare PERIDOT with the machine learning based Ernest approach. Ernest is shown to be effective for predicting the execution time of the applications when combined with a training data selection process [7]. In this study, we specifically focus on the effectiveness of Ernest when trained with limited number of training data, i.e., the two reference executions. Fig. 9 shows the mean error of Ernest with 2 and 4 reference executions. When Ernest is trained with the two reference executions we use for PERIDOT, the mean prediction error of the technique over all applications is 55%. From the figure, the mean error of Ernest can improve when more training data is provided. PERIDOT's accuracy can improve as well with more training data. For example, with 2 additional references, error for $Q26$ goes down from 8.7% to 3.0%. Errors for other applications drop around 1%. However, PERIDOT does not need too many reference experiments to produce accurate predictions and is hence more effective in situations that need quick predictions based on a limited number of executions.

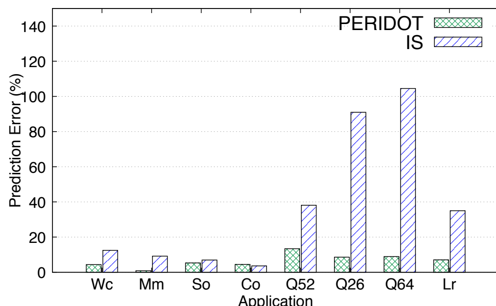


Fig. 8. Prediction errors of PERIDOT and IS

VII. Conclusions and Future Work

Users and operators of a Spark system can benefit from models that provide timely performance predictions for their applications. For example, such models can provide an application developer early insights about scaling behaviour thereby helping them optimize their implementation. They can also be used at runtime by schedulers to size applications with the right amount of resources. Existing approaches require extensive prior executions of applications under a wide range of resource allocations and data sizes of interest. Such data might not be always available. Furthermore, obtaining similar information through controlled experiments can be challenging and even impractical. This paper proposes a technique called PERIDOT that uses just two reference executions on a small subset of the application's input data to obtain a model that can extrapolate its predictions for other input data sizes and resource allocations.

A key aspect that differentiates PERIDOT from existing approaches is that it explicitly models the task wave behaviour and internal dependencies in a Spark application. For any given application, it uses the reference executions to identify application dependencies as a sequence of stage groups. It also computes the times to execute a single wave of tasks in each of the identified stage groups in the reference executions. Modeling the execution time in terms of these parameters ensures that PERIDOT can capture the performance behaviour of an application without the need for exhaustive sampling of the application behavior.

Evaluations based on a diverse set of applications show that PERIDOT can provide accurate predictions. In particular, PERIDOT outperforms competing approaches in predicting the behaviour of applications with complex dependencies. The results also show that PERIDOT requires significantly lesser experimentation effort than machine learning approaches to provide accurate predictions.

Future work will focus on further enhancing PERIDOT's accuracy. In particular, we will focus on alternative methods to model parallel stages and study the impact of those methods on accuracy. Future work will investigate techniques to model how structural properties of data impact execution times. To the best of our knowledge, none of the existing Spark execution time prediction approaches have considered this problem. Finally, we will leverage PERIDOT to realize deadline-driven Spark schedulers as well as optimization frameworks that support job planning exercises.

Acknowledgement

This research is funded by Natural Sciences and Engineering Research Council (NSERC) of Canada and Huawei Technologies Canada Co., LTD. We are grateful to Dr. Shane Anthony Bergsma of Huawei for his thoughtful criticism and feedback. We also thank David Schulz for his support with the ARC cluster.

References

[1] Apache Spark, "Apache spark - lightning-fast unified analytics engine." [Online]. Available: http://spark.apache.org, 2015.

[2] Apache Spark, "Monitoring and instrumentation." [Online]. Available: http://spark.apache.org/docs/latest/monitoring.html.

[3] P. Petridis, A. Gounaris, and J. Torres, "Spark parameter tuning via trial-and-error," in *INNS Conference on Big Data*, pp. 226–237, Springer, 2016.

[4] G. Wang, J. Xu, and B. He, "A novel method for tuning configuration parameters of spark based on machine learning," in *High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (IEEE 18th International Conference on HPCC/SmartCity/DSS)*, pp. 586–593, IEEE, 2016.

[5] G. P. Gibilisco, M. Li, L. Zhang, and D. Ardagna, "Stage aware performance modeling of dag based in memory analytic platforms," in *Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on*, pp. 188–195, IEEE, 2016.

[6] A. Gounaris, G. Kougka, R. Tous, C. T. Montes, and J. Torres, "Dynamic configuration of partitioning in spark applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 7, pp. 1891–1904, 2017.

[7] S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics.," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pp. 363–378, USENIX Association, 2016.

[8] M. T. Islam, S. Karunasekera, and R. Buyya, "dspark: Deadline-based resource allocation for big data applications in apache spark," 2017.

[9] Y. Amannejad, "Peridot experiment log files." [Online]. Available: https://bitbucket.org/YasamanA/sparkdata, 2019.

[10] TPC-DS Benchmark, "Tpc-ds benchmark." [Online]. Available: http://www.tpc.org/tpcds/.

[11] K. Wang and M. M. H. Khan, "Performance prediction for apache spark platform," in *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*, pp. 166–173, 2015.

[12] A. D. Popescu, *Runtime Prediction for Scale-Out Data Analytics*. PhD thesis, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, 2015.

[13] Z. Chao, S. Shi, H. Gao, J. Luo, and H. Wang, "A gray-box performance model for apache spark," *Future Generation Computer Systems*, vol. 89, pp. 58 – 67, 2018.

[14] Y. Amannejad, S. Shah, D. Krishnamurthy, and M. Wang, "Fast and lightweight execution time predictions for spark applications," in *IEEE 9th International Conference on Cloud Computing (CLOUD 2019)*, pp. 1–3, IEEE, 2019.

[15] R. Singhal and P. Singh, "Performance assurance model for applications on SPARK platform," in *Performance Evaluation and Benchmarking for the Analytics Era - 9th TPC Technology Conference, TPCTC 2017, Munich, Germany, August 28, 2017, Revised Selected Papers*, pp. 131–146, 2017.

[16] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni, and S. Rao, "Morpheus: Towards automated slos for enterprise clusters," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, (Savannah, GA), pp. 117–134, USENIX Association, 2016.

[17] F. Pukelsheim, *Optimal design of experiments*, vol. 50. siam, 1993.

[18] M. T. Islam, S. Karunasekera, and R. Buyya, "dspark: Deadline-based resource allocation for big data applications in apache spark," in *13th IEEE International Conference on e-Science (e-Science)*, pp. 89–98, IEEE, 2017.

[19] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, pp. 483–485, ACM, 1967.

[20] University of Calgary, "Advanced research computing cluster." [Online]. Available: https://hpc.ucalgary.ca, 2018.

[21] Abhishek Arora, "Scalable-Matrix-Multiplication-on-Apache-Spark." [Online]. Available: https://github.com/Abhishek-Arora/Scalable-Matrix-Multiplication-on-Apache-Spark, Accessed: Oct. 2018.

[22] D. Ardagna, E. Barbierato, A. Evangelinou, E. Gianniti, M. Gribaudo, T. B. M. Pinto, A. Guimarães, A. P. Couto da Silva, and J. M. Almeida, "Performance prediction of cloud-based big data applications," in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, (New York, NY, USA), pp. 192–199, ACM, 2018.