IEEE Open Journal of the
Computer Society

Received 16 July 2021; accepted 18 August 2021. Date of publication 24 August 2021;
date of current version 16 September 2021. The review of this article was arranged by Associate Editor Dr. Peng Li.

Digital Object Identifier 10.1109/OJCS.2021.3107228

# PERIDOT: Modeling Execution Time of Spark Applications

SARAH SHAH [1], YASAMAN AMANNEJAD [2], DIWAKAR KRISHNAMURTHY [1], AND MEA WANG [3]

[1] Department of Electrical, and Software Engineering, University of Calgary, Calgary, AB T2N 1N4, Canada
[2] Department of Mathematics and Computing, Mount Royal University, Calgary, AB T3E 6K6, Canada
[3] Department of Computer Science, University of Calgary, Calgary, AB T2N 1N4, Canada

Corresponding Author: Sarah Shah (e-mail: sarah.shah1@ucalgary.ca)

This work was supported by the Natural Sciences and Engineering Research Council (NSERC) Canada.

**ABSTRACT** A data analytics application submitted to a Spark cluster often has to finish executing by a specified time target. To use cluster resources effectively, the key challenge is having the ability to gain quick insights on how the execution time of any given 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. Such insights can be used to quickly estimate the required resources and configure a Spark application for a desired execution time using the least amount of resources. Our paper proposes an automated execution time estimation approach called PERIDOT that involves executing a given application under a fixed resource setting with two different-sized, small subsets of its input data to offer fast, lightweight execution time predictions. It analyzes logs from these two executions to estimate the dependencies between internal stages of the application. Information on these dependencies combined with knowledge of Spark's data partitioning mechanisms is used to derive an analytic model that can estimate execution times for other resource settings and input data sizes. Our results from a wide range of applications and multiple Spark clusters show that PERIDOT can accurately estimate the execution time of an application from limited historical data, and suggest the minimum amount of resources required to closely meet an execution time target.

**INDEX TERMS** Apache spark, Big Data processing, performance prediction, performance engineering.

## 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.

Apache Spark supports a resource allocation strategy where users can specify how many cores and memory an application can have. However, specifying the right amount of resources is not trivial. On cluster computing environments, there are typically a large number of choices in terms of the resources that can be assigned to an application. To leverage Apache Spark's resource allocation strategy effectively, the underlying requirement is to have 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, i.e., cores, 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 a tool that can estimate application execution time.

There have been a number of recent efforts at modeling and optimizing the execution time of Spark applications using analytical and machine learning techniques [3]–[18]. These approaches 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. Gathering such history is time consuming and resource intensive, thus not feasible in situations where quick predictions

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

are desired, e.g., to make resource allocation decisions, and extensive access to cluster resources for the experimentation needed to build the models is impractical. Moreover, these techniques have mostly not been verified on a diverse set of applications. In this paper, we aim to reduce the time and resource requirements of the prediction process and offer a generalizable solution for various applications.

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 execution time behaviour of applications.

We evaluate PERIDOT using 13 well-known Spark applications spanning text analytics, linear algebra, machine learning, and Spark SQL executing on two different clusters. Our results indicate that models derived using PERIDOT yield a very good accuracy with a mean prediction error of 7.7% for all 13 applications over a range of resource allocations and input data sizes. In particular, results show that PERIDOT can accurately capture the performance 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 requires over 3,400 core hours, PERIDOT requires only 0.8% of this effort.

We have conducted experiments to evaluate the effectiveness of using PERIDOT for resource allocation exercises. Specifically, we compare our approach with Spark's default resource allocation mechanism, which allocates all available cores in the cluster to an application. The results show that our approach selects more cost-effective resource allocation schemes for any given user defined execution time target. Specifically, our approach results in 43.1% lesser use of cores on average over the baseline technique considered.

The preliminary design and evaluation of PERIDOT is described in our previous conference paper [19]. This paper represents a significant extension in following areas:

1) We have expanded significantly on the preliminary validation presented in our earlier paper through 6,515 core hours of new experiments representing a 260% increase in experimentation effort. Specifically, our validation effort considers 5 additional Spark applications, including applications with more complex internal dependencies. The new experiments also validate PERIDOT's ability to handle multiple task executor resource allocation mechanisms, i.e., executors assigned single and multiple cores (Section VI-B). We have also established the generalizability of PERIDOT by validating it on an additional cluster that has different hardware characteristics (Section VI-E).
2) We have experimentally evaluated the effectiveness of using PERIDOT for allocating resources to Spark applications (Section VI-F).
3) We have fully automated our technique and provided it as a tool for others to use [20].

The remainder of the paper is organized as follows. In Section II, we provide a brief background on Spark. Section III discusses related work. Section IV describes the PERIDOT approach in detail. Section V and VI describe our experiment setup and results, respectively. Conclusions and future work are offered in Section VII.

## 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 can run in sequence or 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 is a language-independent representation of the execution of a Spark application. A Spark application DAG contains a set of vertices and edges, where any given vertex represents an RDD or an input and the outgoing edges represent operations to be applied on the vertex. Fig. 1 is a screenshot from the Spark UI that shows a small part of the Spark DAG for the linear regression application from the Spark standard examples. To see all the stages and jobs of this application, we have provided a simplified visualization of the full DAG in Fig. 2 where rectangles represent jobs and circles represent application stages. This is an application with a simple DAG where all jobs and stages run in sequence.

As mentioned earlier, some applications can produce parallel stages. Fig. 3 shows a small part of the Spark DAG for Query52, a Spark SQL implementation of a query in the TPC-DS [21] benchmark suite. To aid better visualization of the dependencies, Fig. 4 provides our simplified visualization of the DAG with parallel stages. Some stages run in parallel while others run in sequence. Regardless of the size of input data

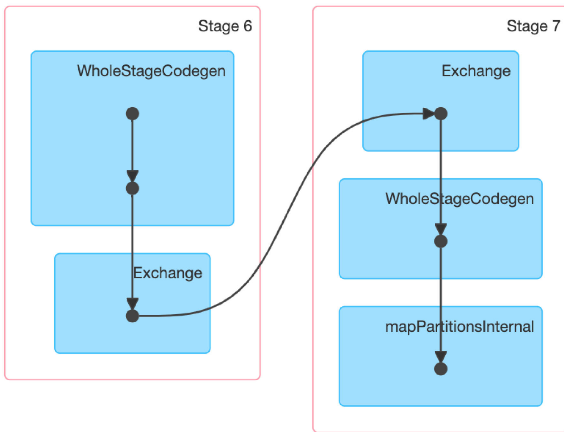



**FIGURE 1. Example of a simple DAG.**

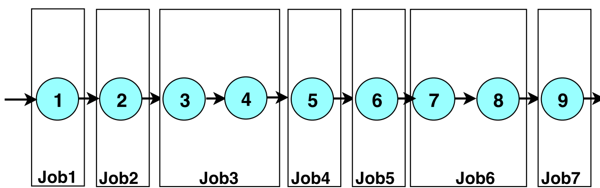



**FIGURE 2. Simplified DAG structure for Linear Regression.**

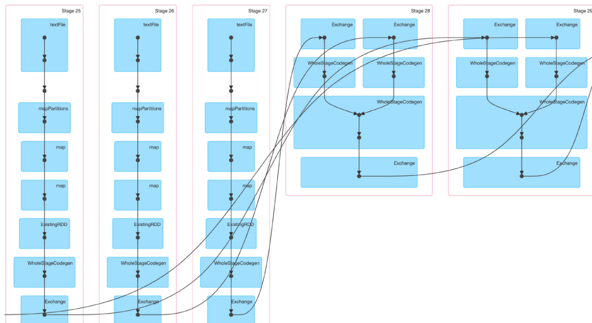


**FIGURE 3. Example of a DAG with parallel stages.**

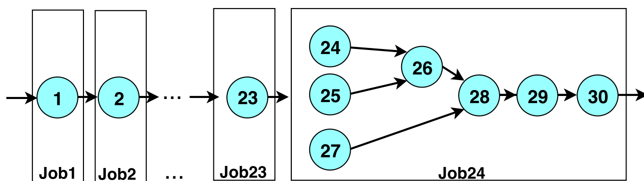
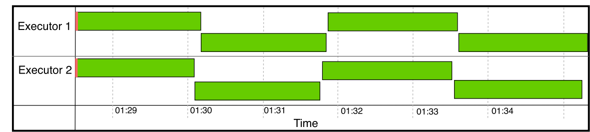



**FIGURE 4. Simplified DAG structure for Query 52.**

or the resource allocation setting, the overall DAG structure remains the same for any given application. As described in Section IV, PERIDOT estimates the dependencies related to jobs and stages 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.

We next focus on behaviour within each stage. To facilitate parallel processing, Spark splits the input data of each stage



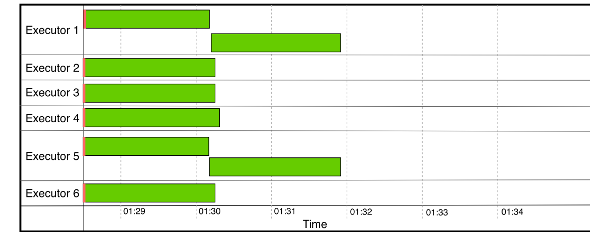




**FIGURE 5. Examples of task waves.**

into smaller data partitions that are typically of equal size. By default, the number of partitions of an input data with size $D$ is $\lceil \frac{D}{block} \rceil$, where $block$ is the block size employed by the underlying file system. However, a stage can override this default partitioning behaviour by explicitly specifying the number of partitions. This is observed in many of the TPC-DS applications in our study.

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 task 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. 5 derived from Spark's visualization of actual stage executions shows two examples of tasks running in waves. In Fig. 5(a), 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. 5(b), 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.

For applications with parallel stages, each wave may consist of tasks from different stages. Fig. 5(b) depicts one such example where tasks from different stages are shown with different colors. From the figure, the individual task execution times in a wave are more heterogeneous than in the single stage scenarios depicted in Fig. 5(a) and Fig. 5(b).

For a given number of executors and with Spark's default partitioning mechanism, increasing the input data size of a stage increases the number of partitions, as the size of partitions generally remains the same. 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, thereby utilizing the available parallelism more effectively.

From the above discussion, the number of waves in a stage and hence the stage execution time generally depends simultaneously on the number of executors and data size. This suggests that the impact of both the number of executors and input size can be modeled by estimating the number of waves and the mean duration of each wave, i.e., wave time, in a stage. We exploit this observation in the next section to develop an analytic *approximation* that captures the execution time of a Spark application.

## III. RELATED WORK

There have been a number of recent efforts on modeling and optimizing the performance of Spark applications [3]–[18]. Singhal and Singh [18] developed a simulation model that captures in detail Spark's internal job processing mechanisms using application's history. 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 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 on 500 execution records where each record contains a list of Spark configuration settings and the execution time achieved with those settings. Gulino *et al.* [5] propose a hybrid approach by modeling individual tasks of a DAG and then offering performance prediction for unseen DAGs. Javaid *et al.* [6] prepare an extensive set of common workload executions and use several Spark features obtained from them to train and compare multiple ML models for execution time prediction. 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.* [22] analyzed workloads on production clusters and showed that about 40% of applications were non-recurrent, i.e., did not have historical execution data. Cheng *et al.* [7] reduce the training samples and lower the model overhead in their Adaboost-based performance prediction model by utilizing projective sampling, however their process still requires a significant amount of training.

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.* [8] 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. Wang and Khan [9] 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.* [10] 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.* [11] 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 [16] 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.* [17] proposed an approach called dSpark for modeling execution time with respect to the number of executors and input data size. They also used dSpark as a conservative resource allocation scheme that allocates resources to applications based on the time needed for their execution as opposed to Spark's default resource scheduling [23] that may allocate all system resources to a job. Similar to Ernest, dSpark uses a set of controlled executions to train a regression model for estimating the execution time of applications. Both these techniques report very good accuracies when there is extensive experiment data available to train the regression models.

Sidhanta *et al.* [24] designed a model called OptEx for estimating Spark application execution times and used it for optimal allocation of cluster resources to applications. OptEx uses a curve fitting technique in combination with analytical expressions. An OptEx model is derived by profiling different categories of applications and extracting parameters for each of these categories that influence performance. Lee *et al.* [25] extended OptEx by including additional parameters to incorporate the probability of failure for an application and used this model to drive resource allocation. Baresi *et al.* [26] proposed a containerized modification of Spark that exploits a black box control theoretic model to realize per-stage deadlines at runtime. Unlike our approach, their approach relies on modifying the default Spark system, which many not be feasible in production environments. In summary, PERIDOT requires lesser experimentation effort than the machine learning approaches discussed in this section.

## IV. EXECUTION TIME MODELING WITH PERIDOT

When users submit their applications to a Spark cluster, they expect to process a certain amount of data within an execution time target, i.e., deadline. Some applications may have tight deadlines, while others may have more flexible deadlines. As described previously, Spark's default resource

allocation method executes all submitted applications in a FIFO order, and each application uses all available cluster resources [23]. This approach prevents resource sharing and can negatively affect applications with tight deadlines. Therefore, Spark users and job schedulers prefer to use a different resource allocation mechanism, e.g., YARN [27], that allows resource sharing. In this mode, Spark users or job schedulers can specify how many executors, cores, and memory an application can have. A fast and accurate execution time prediction model is needed to ensure that users and schedulers request just the right amount of resources so as to meet deadlines in a cost-effective manner. PERIDOT addresses this need.

PERIDOT is a lightweight, analytic execution time estimation approach that does not require exhaustive prior executions of an application to derive a model. Appendix I gives an overview of PERIDOT's modeling process. Given an application, its input data, and the execution time target, 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 four 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 (Section IV-A). Second, we estimate the number of waves that each stage or a group of parallel stages processes and use that to calculate its wave time (Section IV-B). Third, wave times and other parameters extracted from these execution logs are used for estimating the execution time of the application with different input sizes and executor core numbers (Section IV-C).

### 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 detailed information such as stage id, tasks in the stage, the time at which a stage starts, the time at which a task obtains resources and starts executing, and the time at which a task finishes. From the two reference logs, we estimate the dependencies between the stages and characterize how the application's partitioning behaviour changes with data 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. 4, 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 at the same time and hence we create one stage group for them.

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 finish time of the last task in the group and the time when 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. We evaluate the effectiveness of our simplification in Section 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 significantly when the input data size changes. Consequently, the execution time 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. It is essential to use different input data sizes for the two reference executions so that it is possible to identify the fixed and variable stage groups. The identification can be done based on whether there is a change in the number of partitions processed by them, since fixed and variable stage groups contribute to the execution time differently.

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 Section 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

**TABLE 1.** Notations Used in Peridot

| Variable | Description |
|---|---|
| $D$ | Size of input data |
| $N_i$ | Number of waves in stage group $i$ |
| $P_i$ | Number of partitions in stage group $i$ |
| $E$ | Number of executors cores |
| $\hat{W}_i$ | Mean wave time of stage group $i$ |
| $S_i$ | Execution time of stage group $i$ |
| $\hat{P}_i^r$ | Mean number of partitions in references |
| $\hat{D}^r$ | Mean input data size in references |
| $\hat{F}$ | Total duration of fixed stages in references |
| $D^{new}$ | Desired input data size |
| $E^{new}$ | Desired number of cores |
| $N_i^{new}$ | New number of waves in stage group $i$ for desired execution |
| $P_i^{new}$ | Number of partitions in stage $i$ for desired input data |
| $S_i^{new}$ | Duration of stage $i$ in desired execution |
| $T^{new}$ | Predicted execution time for desired execution |

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 Section IV-C.

Table 1 enlists the variables we need to extract using the reference logs obtained in Section IV-A and the variables we need to calculate in order to obtain the execution time for a desired input data size and executor setting. 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.Equation (1) gives us the number of individual sequential execution units, i.e., waves, of execution in each stage group, as described in Section II. The wave time $W_i$ can then be calculated by dividing the total execution time of the stage group $S_i$ 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 $i$.

We note that parallel stage groups can exhibit variability in task execution times since tasks can be from different stages. In this way, our method of calculating wave times for such stage groups in essence averages the execution times of these heterogeneous tasks, thus minimizing the heterogeneity because of the different stages.

$$N_i = \left\lceil \frac{P_i}{E} \right\rceil \tag{1}$$

$$W_i = \frac{S_i}{N_i} \tag{2}$$

We also record as parameters the mean number of partitions $\hat{P}_i^r$ in each stage group of the two references. 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 $F$ 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 $F$. This process is carried out on both reference executions to obtain a mean fixed execution time $\hat{F}$.

### 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 $E^{new}$. Using the parameters obtained from the reference executions, 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. We next estimate the number of waves $N_i^{new}$ using (4). We use (5) to estimate the execution time $S_i^{new}$ of the variable stage group using the mean wave time parameter $\hat{W}_i$ estimated previously. This process is repeated for all variable stage groups and the sum of the $S_i^{new}$ values is added to the fixed partition execution time parameter $\hat{F}$ estimated previously using the reference executions to obtain the execution time prediction $T^{new}$.

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

$$N_i^{new} = \left\lceil \frac{P_i^{new}}{E^{new}} \right\rceil \tag{4}$$

$$S_i^{new} = \hat{W}_i \times N_i^{new} \tag{5}$$

$$T^{new} = \sum_{i \in var} S_i^{new} + \hat{F} \tag{6}$$

PERIDOT is compact since it captures both executor and data scaling using the number of waves as shown by (3) and (4). Modeling in terms of the number of waves and mean time per wave allows PERIDOT to rely on just two reference executions since these references give us the mean wave time as well as the scaling behavior of the variable stages. 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 simplifications with regards to the dependency structure, the characterization of fixed and variable partition stages, and the wave behaviour of stage groups. We experimentally study the impact of such simplifications on accuracy in Section VI.

PERIDOT's prediction process can help to make decisions on the right amount of resources to allocate to an application. Given an application, its input data, and the execution time target, $T^{max}$, PERIDOT uses its prediction process to calculate $T^{new}$ for different executor settings and outputs the minimum number of executors that will meet the execution time target of the application, i.e., $T^{new} < T^{max}$. The results are shown in Section VI-F.

## V. EXPERIMENT SETUP

### A. EXPERIMENT TESTBED

We use PERIDOT to predict the execution time of applications in two different cluster settings and to estimate the number of execution cores required to meet execution time targets. Specifically, we use a multi-node test-bed consisting of nodes from the ARC cluster of the University of Calgary [28]. Our test environment is equipped with Spark 2.2.0 and operates on a common network file system.

### B. EXPERIMENT FACTORS

**Cluster Compute Nodes** - We have evaluated our approach on two clusters consisting of nodes from the ARC system. Our default setting uses nodes from ARC's *Breezy* cluster. In this cluster, each node 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. To show that our model can be generalized to other platforms, we also evaluate it on ARC's *Bigmem* cluster, which uses compute nodes with different characteristics than those of *Breezy*. Each node in this cluster has a four socket Intel Xeon Gold 6148 processor. A socket contains 20 cores each clocked at 2.4 GHz. All cores of one compute node share 3 TB of RAM.

**Applications** - We evaluate our technique with 13 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. Naive Bayes ($Nb$) classifies the input data based on feature values. $Lr$ and $Nb$ are representative machine learning applications that trigger multiple jobs. Correlation analysis ($Co$) calculates the statistical dependency of input variables. While the aforementioned applications are part of the standard Spark distribution, to expand our evaluation, we have also used a Matrix Multiplication ($Mm$) application [29]. $Mm$ receives an $m \times n$ ($m >> n$) matrix, and multiplies the transpose of this matrix with itself to output an $n \times n$ matrix.

To further diversify our applications, we also use Spark SQL queries from the industry-standard TPC-DS benchmark suite [21]. We include these queries because of their more complex DAG structures compared to the rest of the applications. Specifically, we use Query9, Query15, Query26, Query52, Query64, Query70 and Query78. We refer to these queries as $QX$, where $X$ represents the number assigned to each query. $Q9$ is an example of a Spark application that generates parallel jobs. This offers the exploration of a unique scenario in Spark because all other applications under study have serial jobs, which may have parallel stages inside them. To the best of our knowledge, no previous works have explored applications with parallel jobs. $Q15$ is a reporting query that accesses only one table in the TPC-DS data set. $Q26$ and $Q52$ are interactive queries that have been used by other researchers developing performance models [30]. $Q64$ performs complex joins on four different tables to trace patterns in data with respect to selected features. $Q70$ is the most compute-intensive interactive query in TPC-DS suite. $Q78$ is a hybrid query that involves large joins. These queries capture most of the categories discussed in the TPC-DS workload analysis [31], hence providing a well-rounded test-bed for PERIDOT. These applications vary in terms of the number of jobs, number of stages, DAG structures and the operations used.

**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. Each executor in our experiments is configured with 50 GB of RAM, and 5 cores. Each node is configured with 1 executor, so our experiments range from 1 to 8 executors. We have studied other core assignments in Section VI-B where we vary the number of cores assigned to each executor from 1 to 10. 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 [11]. 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 to the number of nodes allocated to the application, a non-linear dependence on nodes that captures inter-node communication overheads, and a linear dependence on nodes.

### C. EXPERIMENT PROCESS

For each experiment in the prediction process, we have an initial phase where 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. Due to the small size of these references, they take a very small time to execute. To evaluate the prediction accuracy, we run each Spark application with all executor core settings and input sizes to collect the application execution times. We repeat each measurement three times and

use the mean of the measured execution times. We compare the measured mean execution times with those predicted by PERIDOT and the other techniques and report the relative error.

To evaluate the effectiveness of PERIDOT's predictions in resource allocation exercises, for each application, we start with an initial execution time target that will help meet a user specified a deadline. We then gradually relax that execution time target up to 200% of the initial target. For each execution time target explored in this manner, we first search the minimum amount of required cores for each application based on PERIDOT's predictions. Then, we run the application with the selected amount of resources and compare the measured execution time of the application with its predetermined execution time target. We compare the resources allocated by PERIDOT with Spark's default resource allocation mechanism which allocates all available resources to the submitted application and observe the resource conservation achieved by PERIDOT.

### D. EXPERIMENT METRICS

To evaluate the accuracy of PERIDOT's predictions, we use the relative error of the predicted execution time values as shown in 7.

$$Error = \left| \frac{T_{predicted} - T_{measured}}{T_{measured}} \right| \times 100 \tag{7}$$

Since we ran each application multiple times in any core-data setting, we are also interested in comparing the 90% Confidence Intervals (CIs) of actual execution time measurements with the predictions. We refer to the lower bound and upper bound of the measured CI as $LB_{measured}$ and $UB_{measured}$, respectively. We define a distance metric $dist$ to quantify the gap between the measured CIs and the predicted values. If the predicted value lies within the CI, then $dist$ is defined to be 0.0. Otherwise, $dist$ is given by (8).

$$dist = Min(|T_{predicted} - UB_{measured}|, \\ |LB_{measured} - T_{predicted}|) \tag{8}$$

Eq. (8) assigns low values to CI and prediction points that almost overlap, i.e., cases where the upper or lower bounds of the measurement are close to the predicted value.

We then define the confidence interval gap $CIG$ using (9). $CIG$ is obtained by normalizing $dist_P$ by the measured application time $T_{measured}$.

$$CIG = \left| \frac{dist}{T_{measured}} \right| \times 100 \tag{9}$$

Finally, to compare the effectiveness of PERIDOT's predictions for selecting resources for an application, we compare the number of cores suggested by our technique to the number of cores assigned by Spark default scheduler and show the resource saving that PERIDOT's predictions offer by selecting the right amount of resources. Using the resources suggested by PERIDOT, applications meet their target deadlines in most experiments. If PERIDOT's measured execution time with the predicted number cores $T_{measured}$ exceeds the execution time target $T_{max}$, we calculate and report an overshoot metric $os$ as shown in (10).

$$os = \frac{T_{measured} - T_{max}}{T_{max}} \times 100 \tag{10}$$

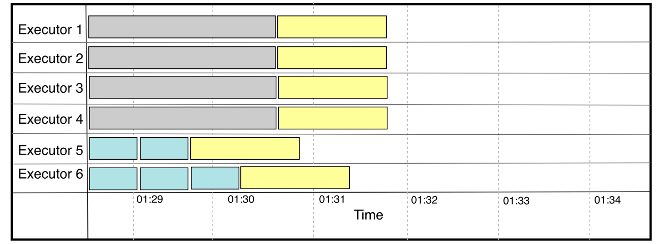
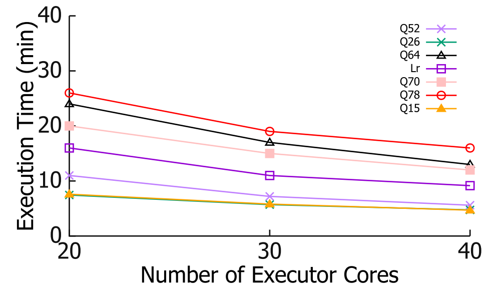



**FIGURE 6.** Examples of task waves in parallel stages.



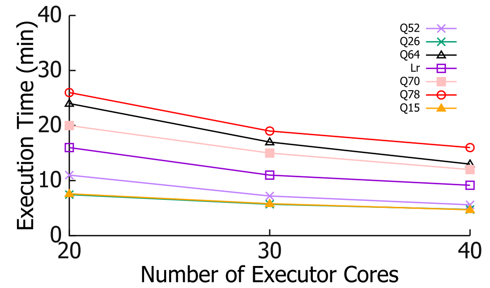




**FIGURE 7.** Execution time of applications.

## VI. RESULTS

This section is organized as follows. We first provide an overview of the performance of the 13 applications we consider in Section VI-A. Then, we evaluate the validity of our prediction approach in Section VI-B by running applications on different number of cores and observing the mean wave times. Section VI-C 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 [20].

### A. APPLICATION PERFORMANCE CHARACTERISTICS

We first show how the execution times of our 13 applications change with the number of executor cores. On our setup, the execution times vary by factors of 3 to 25 while using the executor cores and size settings presented in Section V-B. As

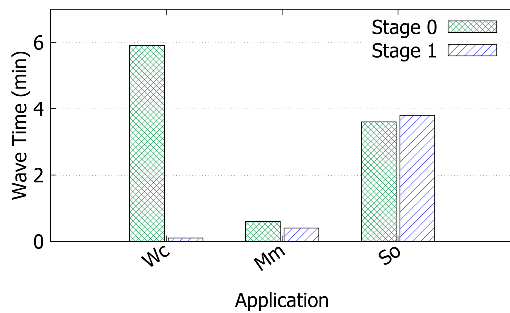



**FIGURE 8.** Wave time of the different stages of the applications.

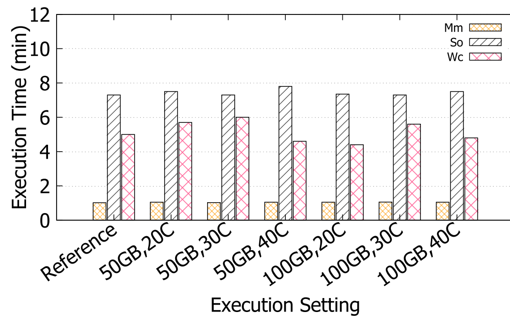



**FIGURE 9.** Wave time of the applications with different settings.

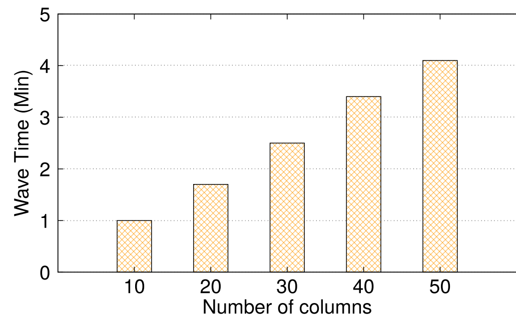



**FIGURE 10.** Effect of number of columns on wave time.

**TABLE 2.** Stage Types Characterization

| | Parallel Stages Groups | 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 |
| **Nb** | 0 | 7 | 8 |
| **Q9** | 1 | 1 | 41 |
| **Q15** | 1 | 1 | 29 |
| **Q26** | 1 | 1 | 30 |
| **Q52** | 1 | 1 | 28 |
| **Q64** | 1 | 1 | 60 |
| **Q70** | 1 | 1 | 32 |
| **Q78** | 1 | 1 | 35 |

shown in Fig. 7, for all applications[1] 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. 8 shows the mean wave times of different stages in $Wc$, $Mm$, and $So$ as measured from the reference executions. From the figure, the mean wave times can vary significantly across stages. Other applications show similar behaviour but we omit the figures since they have a larger number of stages. This trend validates PERIDOT's choice of modeling each stage group separately and aggregating the results of the individual stage group models.

Next, we focus on the wave behaviour of a given variable stage group across different executions. PERIDOT assumes that the prediction for a target execution can be obtained from the wave times 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., 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. 9 plots the wave times of the largest stage of $Mm$, $So$ and $Wc$ under different executor core and data size settings. This figure confirms that the wave time of any given variable stage group is fairly similar across different settings thereby justifying our modeling choice. We observe similar behaviour in other applications including those with parallel stages.

*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. 10 shows that the wave time 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 2 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. 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

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

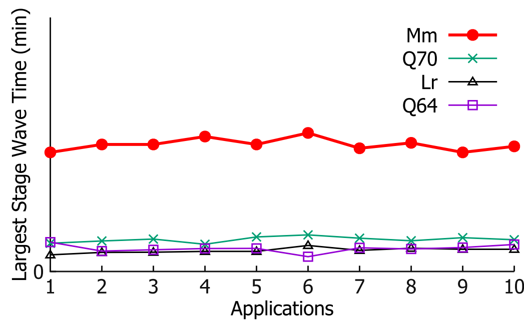



**FIGURE 11.** Wave times under different core assignments.

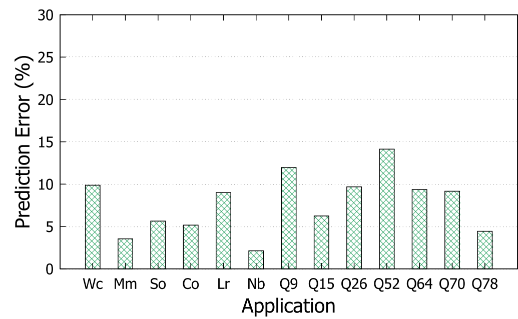



**FIGURE 12.** Prediction errors of PERIDOT on Breezy cluster.

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 $Q26$, $Q64$, and $Q26$, respectively. Although the contribution of fixed stage groups to the execution time is smaller, considering them in the prediction process improves accuracy.

## B. EFFECT OF PROCESSING CORE ASSIGNMENT

PERIDOT uses the mean wave times measured from the reference executions to offer predictions. We now verify whether the number of cores assigned to an executor in these reference executions impacts wave times thereby limiting the ability of PERIDOT to offer predictions for different executor core assignments. We collect the wave times of 4 different Spark applications under various cores assigned to a Spark executor on a single node. We study $Mm$, $Lr$, $Q70$ and $Q64$ to cover different application types and application complexities. We run each application with 2 GB of input data and vary the number of cores from 1 to 10. Fig. 11, demonstrates the similarity in the wave times of the longest stage group of the applications under study under all core settings. We observe this behaviour because the number of cores assigned only changes the number of waves and not the wave time. This experiment verifies that the number of cores used to perform the reference executions does not impact wave times significantly, thereby justifying the use of wave times by PERIDOT for offering predictions.

## C. ACCURACY OF PERIDOT

Fig. 12 shows the mean prediction errors of PERIDOT for the 13 applications on our default cluster, *Breezy*. We perform predictions for all executor and data size settings that are not used by the reference executions. The mean prediction error on the *Breezy* cluster is 7.7%. Many of PERIDOT's predictions are either within or very near the 90% confidence interval of their corresponding measured values. The mean value of the $CIG$ metric is only 4.3% further indicating the effectiveness of the technique.

Further analysis of the errors reveals some general trends. There is no clear correlation between the magnitude of errors and the data size and executor core settings. Applications that had a simple DAG structure with no parallel stages resulted in a small mean error of 5.0% because they have very well defined fixed and varying stage groups in them. In contrast, Spark SQL applications, which have complex DAGs and parallel stages, show a slightly higher mean error of 9.3%. In our experiments, $Q52$ has the highest mean error among all applications at 14.1%. The main reason behind the higher errors is that PERIDOT assumes that a stage group where the number of partitions stays the same across the reference executions can be modeled as a fixed stage group whose execution time is a constant across different runs. However, this is not strictly true for Spark SQL applications. Despite not displaying an increase in the number of partitions with an increase in data size, these stage groups show small increase in the execution time because of an increase in the size of data processed by each partition. Capturing the exact behaviour of such stages requires more complex models, which we will explore as future work.

We note that increasing the number of reference executions may provide better averaged values of our model parameters and consequently can improve prediction accuracy. However, this increases the time needed for executing the references. We observed that the improvement in the accuracy beyond 2 reference executions is negligible. For example, increasing the number of reference executions from 2 to 4 for the Correlations (Co) application decreases the prediction error by only 2%, while doubling the time spent on reference executions. Thus, 2 references allow a good trade off between accuracy and quickness. If cluster operators wish to increase the number of references, this can still be done with no change in our proposed method.

PERIDOT provides quick predictions. For instance, exhaustive experimentation of all possible configurations we explore for all 13 applications requires over 3,400 core hours. The core hours needed by PERIDOT are only 0.8% of that incurred by exhaustive experimentation. We also note that the time taken to analyze the log files of application executions in order to extract the model parameters is negligible.

## D. COMPARISON WITH OTHER APPROACHES

Fig. 13 compares PERIDOT and Ideal Scaling (IS). PERIDOT significantly outperforms IS offering predictions that are 10 times more accurate overall than IS. For the $Wc$, $Mm$, $So$,

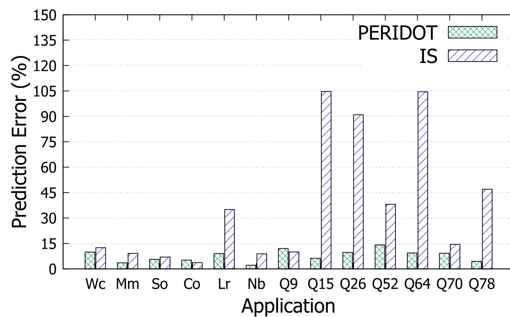



**FIGURE 13. Prediction errors of PERIDOT and IS.**

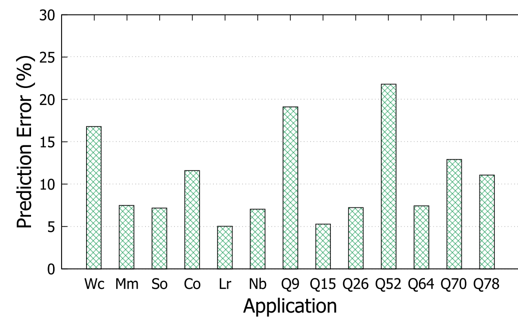



**FIGURE 14. Prediction errors of PERIDOT on Bigmem cluster.**

*Lr* and *Co* applications, the mean accuracy of PERIDOT is 2 times that of IS for the *Breezy* experiments. IS performs extremely poor with Lr and Spark SQL applications, which have complex DAGs. For instance with *Q*64, which has a very complicated DAG, the average prediction error of IS reaches 104% in the *Breezy* experiments. Other SQL queries show average prediction errors of up to 116% with the IS prediction method. These results show that the behaviour of applications with complex DAGs and parallel stages cannot be predicted by mere linear extrapolation of execution time.

Finally, we compare PERIDOT with the state-of-the-art 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 [11]. In this study, we specifically focus on the effectiveness of Ernest when trained with limited amount of training data, i.e., the two reference executions. Ernest has a mean error of 20.4% over the 13 applications used in this study with the maximum error reaching up to 60%. The mean error of Ernest is nearly 2.6 times of PERIDOT's mean error. The mean error of Ernest can improve when more training data is provided. Using 4 reference executions the mean error of Ernest reduces, however it is still 1.8 times of PERIDOT's mean error. This confirms that PERIDOT is more effective in situations that need quick predictions based on a limited number of executions.

### E. EFFECT OF CLUSTER NODES

To explore how our modeling techniques can be generalized, we evaluated our model on the *Bigmem* cluster [28]. Each node in this cluster has a very different resource setting than the *Breezy* nodes. We run the same set of experiments by varying the input data and core sizes and record the measured execution times. We then compare them against the predictions of PERIDOT. On *Bigmem*, we achieved the mean error and mean *CIG* of 10.8% and 9.7%, respectively over all 13 applications. Considering both clusters and all applications, PERIDOT achieves a mean error and mean *CIG* of 9.1% and 6.9%, respectively. These results validate the modeling choices described in Section IV.

### F. USING PERIDOT FOR RESOURCE ALLOCATION

In this section, using an example, we show that PERIDOT's execution time predictions can be used for selecting the minimum amount of resources while closely meeting application deadlines. To evaluate the effectiveness of PERIDOT for resource allocation, we compare it against the default resource allocation of Spark, which assigns all available cores to the submitted application. For each application at 50 GB input size, we start with an execution time target, and gradually relax that target. We assume our system to have a maximum of 50 execution cores that can be assigned to any application. To make the comparison easier, we start experiments with the execution time target that can be met with 50 cores. In this case, both the default resource allocation and PERIDOT will assign 50 cores to the application. Then, we conduct 10 additional experiments for each application, where we relax the execution time target of the application from 20% to 200% of the initial execution time target.

Fig. 15 shows the number of cores allocated by the default static approach and number of cores selected by PERIDOT for 4 applications. Spark's default resource allocator will statically assign all of the 50 cores to the application regardless of its execution time target. PERIDOT adjusts the resources based on its prediction for the execution time of the application under different core configurations. We record the actual execution times for each application with 50 GB of input data over core settings from 10 to 50 in steps of 5 on the Bigmem cluster. For each experiment, we show the actual minimum number of required cores for meeting the execution time targets based on the measured execution times of the applications. From the figure, the cores selected by PERIDOT closely follow the number of cores that are actually required. Similar behaviour is observed with the rest of the applications in our experiment set. Considering all applications and scenarios we explore, PERIDOT's resource allocation provides 43.1% conservation of resources on average over the default approach.

The execution time targets may be violated when PERIDOT selects less cores than required. In case of a violation, we record the amount of the overshoot percentage *os*. Considering all applications and the 10 time targets that we study

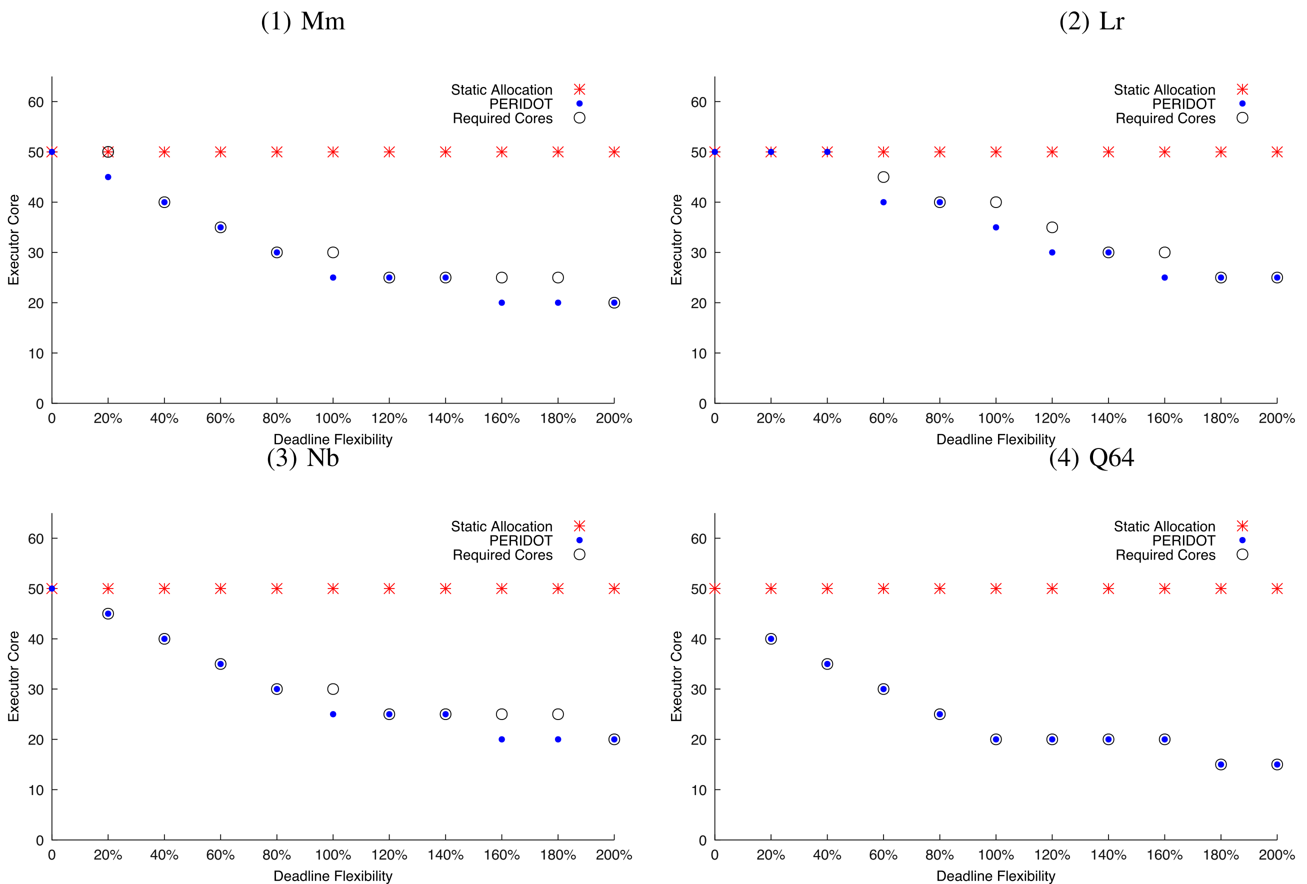



**FIGURE 15.** Executor cores allocated with the default approach, PERIDOT, and actual required cores.

**TABLE 3.** Resource Savings & Violations

| # of cores | Overshoot $os$ | Resource conservation (%) |
|---|---|---|
| 50 | 2.9 | 0% |
| 45 | 7.8 | 10% |
| 40 | 10.6 | 20% |
| 35 | 16.8 | 30% |
| 30 | 20.8 | 40% |
| 25 | 34.8 | 50% |
| 20 | 53.2 | 60% |

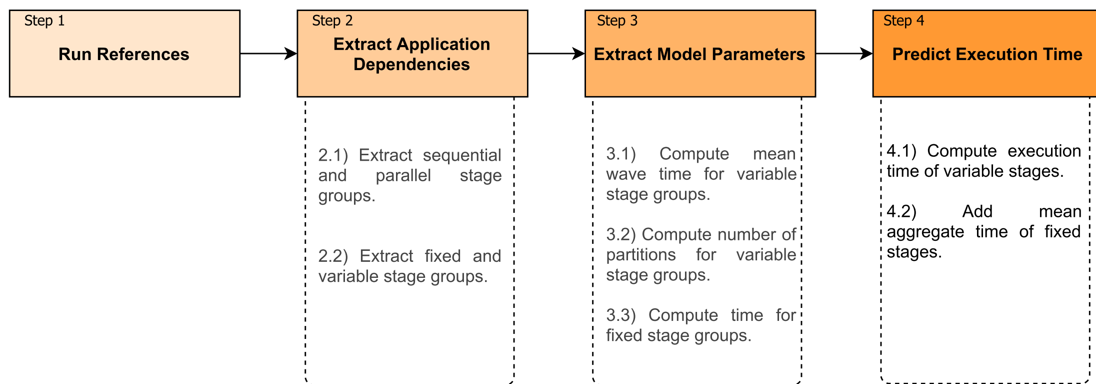
for each application, the mean $os$ is 8.3%. This is a reasonable value compared to the 43.1% resource conservation of PERIDOT.

Table 3 compares other static allocation strategies in terms of execution time target violations and resource conservation. As expected, assigning lower number of cores realizes resource savings but results in time target violations. The highest resource conservation relative to the 50 core assignment is 60% and is achieved by statically allocating 20 cores. However, this policy results in the highest execution time target violation of 53.2%. In contrast, PERIDOT offers 43.1% resource conservation relative to the 50 core case at the cost of only an 8.3% target time violation. These results offers a good balance of resource conservation and time target violation compared to static allocation techniques.

## 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 breaks down Spark to it's most basic operative functionality and 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 variable 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. Our selected set of applications cover a wide range of Spark operations and internal dependencies. PERIDOT outperforms competing approaches. In particular, it is more effective in capturing the execution time 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. We also show that PERIDOT is highly effective for right sizing a Spark application such that the application satisfies a specified deadline while using the least amount of resources.

Future work will focus on further enhancing PERIDOT's accuracy. In particular, we will focus on alternative methods to model stages that have fixed partition size but varying input record size and study the impact of those methods on accuracy.

## APPENDIX I. STEPS INVOLVED IN PERIDOT

The flowchart at the top of the page shows the steps involved in the prediction process of PERIDOT. The main steps are shown in colored boxes and the tasks involved in each step are shown in white boxes.

## REFERENCES

[1] Apache Spark, "Lightning-fast unified analytics engine," 2018.
[2] Apache Spark, "Monitoring and instrumentation," 2017.
[3] P. Petridis, A. Gounaris, and J. Torres, "Spark parameter tuning via trial-and-error," in *Proc. INNS Conf. Big Data*, Springer, 2016, pp. 226–237.
[4] G. Wang, J. Xu, and B. He, "A novel method for tuning configuration parameters of spark based on machine learning," in *Proc. 18th Int. Conf. High Perform. Comput. Commun.; IEEE 14th Int. Conf. Smart City; IEEE 2nd Int. Conf. Data Sci. Syst.*, 2016, pp. 586–593.
[5] A. Gulino, A. Canakoglu, S. Ceri, and D. Ardagna, "Performance prediction for data-driven workflows on apache spark," in *Proc. 28th Int. Symp. Modeling, Anal., Simul. Comput. Telecommun. Syst.*, 2020, pp. 1–8.
[6] M. U. Javaid, A. A. Kanoun, F. Demesmaeker, A. Ghrab, and S. Skhiri, "A performance prediction model for spark applications," in *Proc. Int. Conf. Big Data*, Springer, 2020, pp. 13–22.
[7] G. Cheng, S. Ying, B. Wang, and Y. Li, "Efficient performance prediction for apache spark," *J. Parallel Distrib. Comput.*, vol. 149, pp. 40–51, 2021.
[8] G. P. Gibilisco, M. Li, L. Zhang, and D. Ardagna, "Stage aware performance modeling of dag based in memory analytic platforms," in *Proc. IEEE 9th Int. Conf. Cloud Comput.*, 2016, pp. 188–195.
[9] K. Wang and M. M. H. Khan, "Performance prediction for apache spark platform," in *Proc. 17th Int. Conf. High Perform. Comput. Commun., IEEE 7th Int. Symp. Cyberspace Safety Secur., IEEE 12th Int. Conf. Embedded Softw. Syst.*, pp. 166–173.
[10] A. Gounaris, G. Kougka, R. Tous, C. T. Montes, and J. Torres, "Dynamic configuration of partitioning in spark applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 7, pp. 1891–1904, Jul. 2017.
[11] S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *Proc. 13th USENIX Symp. Networked Syst. Des. Implementation*, 2016, pp. 363–378.
[12] M. T. Islam, S. Karunasekera, and R. Buyya, "dSpark: Deadline-based resource allocation for big data applications in apache spark," in *Proc. IEEE 13th Int. Conf. E-Sci.*, 2017, pp. 89–98.
[13] A. D. Popescu, Runtime prediction for scale-out data analytics. Ph.D, thesis, Dept. Comput. Sci., École Polytechnique Fédérale de Lausanne, 2015.
[14] Z. Chao, S. Shi, H. Gao, J. Luo, and H. Wang, "A gray-box performance model for apache spark," *Future Gener. Comput. Syst.*, vol. 89, pp. 58–67, 2018.
[15] Y. Amannejad, S. Shah, D. Krishnamurthy, and M. Wang, "Fast and lightweight execution time predictions for spark applications," in *Proc. IEEE 9th Int. Conf. Cloud Comput.*, 2019, pp. 1–3.
[16] F. Pukelsheim, *Optimal Design of Experiments*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1993.
[17] M. T. Islam, S. Karunasekera, and R. Buyya, "dSpark: Deadline-based resource allocation for Big Data applications in apache spark," in *Proc. 13th IEEE Int. Conf. e- Sci.*, 2017, pp. 89–98.
[18] R. Singhal and P. Singh, "Performance assurance model for applications on SPARK platform," in *Proc. Technol. Conf. Perform. Eval. Benchmarking*, Munich, Germany, 2017, pp. 131–146.
[19] S. Shah, Y. Amannejad, D. Krishnamurthy, and M. Wang, "Quick execution time predictions for spark applications," in *Proc. Int. Conf. Netw. Service Manage.*, 2019, pp. 1–9.
[20] S. Shah, Y. Amannejad, D. Krishnamurthy, and M. Wang, "Peridot Data Set," 2020. [Online]. Available: https://github.com/Yasaman-A/Peridot
[21] TPC-DS Benchmark, " Tpc-Ds Benchmark," [Online]. Available: http://www.tpc.org/tpcds/
[22] S. A. Jyothi *et al.*, "Morpheus: Towards automated slos for enterprise clusters," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, Savannah, GA, USA, 2016, pp. 117–134.
[23] Apache Spark, "Job Scheduling," 2017.
[24] S. Sidhanta, W. Golab, and S. Mukhopadhyay, "Optex: A deadline-aware cost optimization model for spark," in *Proc. 16th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, 2016, pp. 193–202.
[25] J. Lee, B. Kim, and J.-M. Chung, "Time estimation and resource minimization scheme for apache spark and hadoop big data systems with failures," *IEEE Access*, vol. 7, pp. 9658–9666, 2019.

[26] L. Baresi, A. Leva, and G. Quattrocchi, "Fine-grained dynamic resource allocation for Big-Data applications," *IEEE Trans. Softw. Eng.*, vol. 47, no. 8, pp. 1668–1682, Aug. 2021.
[27] Apache Spark, "Running Spark on YARN," 2017.
[28] ARC, "Research Computing Cluster," 2018. [Online]. Available: https://hpc.ucalgary.ca
[29] A. Arora, "Scalable-Matrix-Multiplication-on-Apache-Spark," [Online]. Available: https://github.com/Abhishek-Arora/Scalable-Matrix-Multiplication-on-Apache-Spark, Accessed: Oct. 2018.
[30] D. Ardagna *et al.*, "Performance prediction of cloud-based Big Data applications," in *Proc. ACM/SPEC Int. Conf. Perform. Eng.*, New York, NY, USA, 2018, pp. 192–199.
[31] M. Poess, R. O. Nambiar, and D. Walrath, "Why you should run TPC-DS: A workload analysis," *VLDB*, vol. 7, pp. 1138–1149, 2007.

**SARAH SHAH** received the B.Sc. degree from the National University of Sciences and Technology, Pakistan, and the M.Sc. degree from the University of Calgary. She is currently the Ph.D. student with University of Calgary. Her research interests include software performance engineering for big data analytics systems.

**YASAMAN AMANNEJAD** received the bachelors and masters degrees in computer information technology from the Amirkabir University of Technology and the Ph.D. degree in software engineering from the University of Calgary. She is an Assistant Professor with Mount Royal University. She received the Ph.D. degree in software engineering from the University of Calgary, Canada. Her research interests include software engineering, data analytics, and machine learning.

**DIWAKAR KRISHNAMURTHY** received the bachelors degree in electrical and electronics engineering from the Thiagarajar College of Engineering, Madurai, India in 1995 and the M.Eng. and PhD. degrees in electrical engineering from Carleton University, Ottawa, Canada in 1998 and 2004, respectively. He is a Professor with the University of Calgary. His research interests focused on the performance evaluation of software systems. He is currently involved in research projects related to cloud computing, virtualization technologies, big data analytics, and healthcare simulation.

**MEA WANG** received the Bachelor of Computer Science (Honours) degree from the University of Manitoba, Winnipeg, Manitoba in 2002. She studied in the Department of Electrical and Computer Engineering at the University of Toronto from 2002 to 2008 and earned the MASc and PhD degrees in 2004 and 2008, respectively. She is an Associate Professor with the Department of Computer Science, University of Calgary. Her research interests include peer-to-peer networking, multimedia networking, cloud computing, as well as networking system design and development.