2020-12-16

Performance Modeling for Spark Applications

Shah, Sarah

UNIVERSITY OF CALGARY

Performance Modeling for Spark Applications

by

Sarah Shah

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING

CALGARY, ALBERTA

DECEMBER, 2020

Abstract

The Apache Spark cluster computing platform is being increasingly used to execute big data applications. This research explores two distinct but related problems pertaining to the performance of Spark applications.

The first problem focuses on predicting the execution time of a Spark application. Predicting the execution time of an application before submitting it to the cluster will allow a user to quickly estimate the right amount of resources required to achieve a desired execution time target. Most existing prediction techniques require extensive historical executions of an application, thus being time consuming and requiring extensive cluster resources, making it hard for them to be deployed in real-world settings. I address this problem by proposing a quick and lightweight analytic execution time prediction technique that only requires two reference executions of any given application to offer predictions. I show that the proposed technique provides accurate predictions and outperforms other baselines I consider.

The second problem considers Spark environments where applications encounter interference, i.e., contention for system resources shared with other applications. Spark operators often co-locate multiple applications on cluster nodes to improve resource utilization. However, this can lead to interference thereby adversely impacting an application's execution time. Several studies have proposed models to detect and diagnose such interference. However, these models require extensive historical training data to be effective and can take considerable time to offer predictions. I conduct a systematic study to devise a machine learning (ML) based technique that can diagnose interference quickly and accurately without requiring extensive training data. Specifically, I explore techniques that would allow the model to generalize to scenarios not captured in the training data, e.g., unseen applications and input data sizes, and to quickly offer online predictions without waiting for an interfered application to complete.

Acknowledgements

I want to start by first thanking my supervisors Dr. Diwakar Krishnamurthy and Dr. Yasaman Amannejad for their never-ending support and patience during the course of my research. Their indispensable guidance has been extremely helpful in giving me new perspectives on all the problems I addressed in my thesis, and educating me in delivering state-of-the-art research. I am very grateful for being given the opportunity to work under them. I owe the completion of this thesis to their dedicated efforts in making a good researcher out of me.

Next, I want to thank my committee members Dr. Roman Shor and Dr. Gias Uddin for taking the time to review my thesis and attend my defense. I also want to thank the Graduate Program Administrator, Ella Lok, for always being generously helpful in completing all the administrative tasks related to my graduate studies. I would like to appreciate David Schulz as well for his swift and consistent support on the ARC cluster that I used heavily for my experimentation.

Lastly, I want to express my gratitude to my parents who have been a constant source of support and hope for me and have enabled me to reach this far in my academic career, and to all my friends and lab mates who have been supportive throughout my degree and offered help in getting through the hurdles in my research.

Table of Contents

List of Tables

List of Figures and Illustrations

List of Symbols, Abbreviations and Nomenclature

Abbreviations

AutoML	Automatic Machine Learning
CI	Confidence Interval
DAG	Directed Acyclic Graph
GBDT	Gradient Boost Decision Trees
GC	Garbage Collection
GCP	Google Cloud Platform
IS	Ideal Scaling
JVM	Java Virtual Machine
ML	Machine Learning
RDD	Resilient Distributed Dataset
RQ	Research Question
SoI	Source of Interference
TPC-DS	TPC Decision Support
VM	Virtual Machine

Nomenclatures

os	Amount of overshoot in case of missed execution time targets
$block$	Block size of the file system
CIG	Confidence Interval Gap
D^{new}	Desired input data size
E^{new}	Desired number of cores
$dist$	Distance of the estimated application time from the measured CI
S_i^{new}	Duration of stage i in desired execution
$LB_{measured}$	Lower bound of measured CI
$\hat{D}^r$	Mean input data size in references
$\hat{P}_i^r$	Mean number of partitions in references
$\hat{W}_i$	Mean wave time of stage group i
$T_{measured}$	Measured execution time of an application
E	Number of executors cores
P_i	Number of partitions in stage group i
P_i^{new}	Number of partitions in stage i for desired input data
N_i	Number of waves in stage group i
N_i^{new}	New number of waves in stage group i for desired execution
$Error$	Relative error of the predicted and actual execution times
S_i	Execution time of tage group i
D	Size of input data
t	Prediction threshold
$T_{predicted}$	Predicted execution time of an application
T_{max}	Target execution time of an application
T^{new}	Total duration of execution in desired execution

$\hat{F}$	Total duration of fixed stages in references
$UB_{measured}$	Upper bound of measured CI
W_i	Wave time of stage group i

Chapter 1

Introduction

1.1 Motivation

The Apache Spark cluster computing platform [1] is being increasingly used to develop big data analytics applications. Spark supports the development of a wide range of data-intensive applications spanning areas such as text analysis, decision support, ML, graph analytics, and stream processing. Performance 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. Long execution times that go beyond this target can frustrate end users and can cause them to discontinue using the cluster. Cluster operators, on the other hand, would like to provision resources efficiently to applications such that application execution time targets are satisfied while simultaneously maximizing utilization of cluster resources. Maximizing resource utilization allows operators to reduce the overall costs of providing their service. This thesis considers two inter-related research problems associated with the performance of Spark applications.

The first problem I consider is predicting the execution time of a Spark application. 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. For example, a prediction tool can help users and operators determine just the minimum amount of resources, e.g, processing cores, needed to meet an application's execution time target. This can in turn facilitate efficient usage of cluster resources. While Spark provides sophisticated tools to monitor application performance [2], it does not yet support such an execution time prediction tool.

There have been a number of recent efforts at modeling and optimizing the execution time of Spark applications using analytical and ML techniques [3–9]. 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, e.g., to make resource allocation decisions, and extensive access to cluster resources for the experimentation needed to build the models is impractical. This motivates the *first objective* of my thesis, which is to design an execution time prediction technique that can offer quick predictions without the need for extensive cluster resources and experimentation to build the underlying performance model.

The second problem I investigate is predicting the execution time of a Spark application in environments where multiple applications can share a cluster resource. To increase cluster utilization, operators often configure Spark such that multiple co-located applications can simultaneously share the resources, e.g., processing cores and memory, of a cluster node. Sharing of resources can also happen in cloud-based clusters where Spark Virtual Machines (VMs) might be co-located with other VMs on physical machines. With resource sharing, applications can compete with each other for shared node resources, thereby interfering with each other's performance. Such performance interference can cause a Spark application to fail or take a very long time to execute. If unmitigated, interference can cause frustration to cluster users and lead to inefficient use of cluster resources. This motivates the *second objective* of this thesis, which is to develop an automated technique that can detect interference in a timely manner and diagnose the root cause of the interference, i.e., identify the node resource experiencing the most contention. Such a technique can allow a cluster operator to terminate an interfered application quickly and launch it on another cluster node where

interference for the resource is less likely.

Although existing studies have focused on interference in Spark clusters [10–16], there are several open issues that still remain to be addressed. For example, while many of these techniques can detect interference, they are not designed to isolate the root cause, especially at the granularity of a node resource. This can impair the ability of a user or operator to effectively mitigate the interference. Furthermore, many of the existing approaches train specific ML models for each Spark application of interest by executing each application under a wide range of input data sizes, resource allocation settings, and interference conditions. Obtaining such extensive training data for every application submitted to the cluster can be time consuming and can demand considerable cluster resources. This will in turn impact the timeliness and feasibility of mitigating interference. Moreover, most of the existing techniques are designed to analyze interference *after* an application has finished its execution. This lack of agility in detecting and mitigating interference can be problematic for long running applications. This motivates the need for techniques that can provide insights about interference while an application is executing. Finally, existing work has been validated only on a limited set of Spark applications. My research addresses these open issues by systematically investigating how to construct an ML based model that will offer accurate predictions quickly, require very little training data, and generalize to scenarios not captured in the training data.

1.2 Research Challenges and Questions

In this section, I outline the key challenges that need to be addressed in each of the problems I consider. I use these challenges to describe a set of concrete research questions related to each problem.

1.2.1 Quick Execution Time Predictions for Spark Applications

As discussed previously, cluster users and operators can benefit from quick predictions of how long any given application will take to execute under various resource allocations, e.g., number of processor cores assigned to the application, and data sizes of interest. Such estimates would help configure an application's execution such that execution time targets are met with just the right amount of cluster resources. The key challenge I consider is developing a prediction technique that does not require extensive past execution data of an application covering the range of resource allocations and data sizes of interest. Gathering such extensive data can be time consuming and resource intensive and thus not practical in a production cluster setting. Furthermore, production clusters encounter a large number of new applications [17], rendering techniques relying on the existence of extensive past application executions ineffective. To address this challenge, I focus on answering the following research questions:

- **RQ 1-1:** Can I leverage knowledge about an application's internal structure and the Spark platform to offer execution time predictions with only limited experimentation effort?
- **RQ 1-2:** Can this prediction technique offer accurate predictions for applications of different structures, resource demands, resource allocations, and data sizes?
- **RQ 1-3:** How does this technique compare in terms of accuracy and experimentation effort with existing approaches based on extensive data collection?
- **RQ 1-4:** How effective is the prediction technique in identifying the right amount of cluster resources required to achieve various application execution time targets?

1.2.2 Diagnosing Performance Interference in Spark Applications

Diagnosing performance interference requires addressing the following challenges:

- The training effort to build an interference diagnosis model should be kept low. A Spark cluster is likely to encounter different types of applications, a variety of data sizes, and interference caused due to contention for various types of node resources, e.g., processors and memory. Building a model that can offer accurate predictions for such a wide array of scenarios might require extensive training, which is not feasible in real-life clusters. This necessitates a technique for the development of a model that can be trained quickly with little effort and can provide predictions for scenarios not part of the training data.

- Diagnosing the root cause of interference, i.e., identifying the node resource undergoing the most contention, can be a challenge. For example, consider the problem of distinguishing between processor and memory contention at a cluster node. Both types of bottlenecks can lead to high processor utilization thereby making it a challenge to diagnose the root cause. Furthermore, in some environments it might not be possible to obtain complete information on how a node's resources are utilized. For example, consider a Spark cluster hosted on a public-cloud platform. While the operator of such a cluster can obtain resource usage pertaining to their own applications, they do not have visibility into how other extraneous applications are using the shared cloud resources. Interference diagnosis is even more challenging in such settings.

- For long running applications that consume significant cluster resources, interference detection and diagnosis needs to be quick. Slow diagnosis will lead to significant user frustration and inefficient use of cluster resources. Conse-

quently, predictions need to be made *while* the application is still executing. Thus, the technique should be robust enough to offer accurate predictions even with partial information about an application's execution.

I consider the following research questions to address these challenges:

- **RQ 2-1:** How should an ML model be designed so that it can offer accurate interference diagnosis without requiring extensive training data?
- **RQ 2-2:** How well does the ML model generalize to resource allocation and data size settings that ar not part of the training data?
- **RQ 2-3:** How feasible is it to build models for scenarios where resource usage information is not available?
- **RQ 2-4:** How well does the ML model generalize to applications that are not part of the training data?
- **RQ 2-5:** Can the ML model diagnose interference problems quickly while an application is still executing?

1.3 Contributions

1.3.1 Peridot

My first contribution is 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 input datasets, Peridot significantly reduces the time and resources required to construct application models, thereby facilitating quick insights into the execution time behaviour of applications. Peridot's fast execution time estimations can be used to quickly configure any given Spark application submitted to a cluster for a desired execution time target. Moreover, Peridot can be implemented in 3 simple steps and requires very little information and calculations to extract the execution time prediction, thus offering a lightweight solution. Peridot is discussed in detail in Chapter 4. I have automated Peridot and made it available online [18] along with all execution logs and experiment data used in this thesis.

Experimental results from a wide variety of Spark applications deployed on two different real Spark clusters indicate the effectiveness of Peridot thereby answering **RQ 1-1** and **RQ 1-2**. My experiments indicate, as expected, that naive prediction techniques that assume ideal application speedup yield poor accuracy. In contrast, models derived using Peridot yield a very good accuracy with a mean prediction error of 9.3% for all applications over a range of resource allocations and input data sizes (**RQ 1-2**). Generating models and predictions using Peridot is quick. For instance, while exhaustive experimentation of all possible configurations I explore requires over $3,400$ core hours, Peridot requires only 0.8% of this effort.

I compare Peridot with a state-of-the-art ML based prediction approach [8] that is agnostic to an application's internal data dependencies and partitioning mechanisms. Specifically, I compare the ability of this technique and Peridot to offer accurate predictions with limited experiment data. In contrast to Peridot, the ML technique yields poor predictions when only limited training data is available (**RQ 1-3**). These results suggest that Peridot is feasible and effective in offering quick, resource-efficient, and accurate predictions.

I also evaluate the effectiveness of using Peridot for resource allocation exercises. Specifically, I compare my approach with Spark's default resource allocation mechanism, which

statically allocates all available cores in the cluster to an application. I vary available cores in the experiments to compare these two techniques under various scenarios. The results show that my approach selects more cost-effective resource allocation schemes for any given user defined execution time target. Specifically, my approach results in 43.1% lesser use of cores on average over the baseline technique considered (**RQ 1-4**).

1.3.2 Diaspore

My second contribution addresses the open issues related to the interference diagnosis problem outlined in Section 1.2. I address these issues by systematically investigating how to construct an ML based model that requires very little training data and can generalize to scenarios not captured in the training data. Specifically, I develop Diaspore, an ML based technique that can detect interference and identify the cluster node resource responsible for the interference. I focus on differentiating interference due to contention for processor cores from that due to contention for memory. This represents a stress case for my technique since contention for memory can lead to high processor utilization and can therefore be mistaken for processor contention.

Diaspore only requires training experiments for each application of interest at small scale (**RQ 2-1**), i.e., small input data sizes and cores, in a sandbox environment within the cluster. In the training executions, the type, i.e., processor or memory, and magnitude of interference are varied. Diaspore collects application metrics, e.g., execution times, and system metrics, e.g., processor utilization, from these executions. A custom feature engineering technique is used to scale metrics collected from an interfered execution with the corresponding metrics from a non-interfered execution. Data transformed in this manner is used within an existing Automated Machine Learning (AutoML) system [19] to derive a single global model that can classify whether an execution of any given application is suffering due to interference and if so, the type of contention, i.e., processor or memory, driving the interference.

In Chapter 5, I provide a comprehensive validation of Diaspore that considers a diverse

set of 13 Spark applications executing on a real cluster. My results provide several novel insights into the design of an ML based model for interference diagnosis. Unlike existing approaches, results show that Diaspore achieves good accuracy even when it is trained only with small scale executions (**RQ 2-2**). Specifically, it provides an accuracy of 83.3% when training data derived from 1 GB and 2 GB datasets, executed on 1 and 2 node configurations, is used to predict for larger scale executions. Using the larger scale executions to train the ML model yields an accuracy of 100.0% but incurs 45.0% more training time and 90.0% additional training resources, i.e., cluster cores.

My experiments also provide insights on the feasibility of offering predictions in environments where system metrics that capture resource usage are not available. Specifically, results show that Diaspore can offer accurate predictions in this scenario when it is trained with both small and large scale data (**RQ 2-3**).

I also validate the generalizability of Diaspore for applications. Results show that my feature scaling approach allows Diaspore to generalize to applications not present in the training data. I observe an average accuracy of 83.4% in experiments where I exclude one of the 13 applications during training and use the resulting model to predict for the excluded application (**RQ 2-4**). This result shows that my technique can save on training time and provide timely predictions in production settings, which typically encounter a significant fraction of non-recurring, i.e., new, applications [17]. I have made the data and the scripts used in developing Diaspore online [20].

Finally, results show that Diaspore can offer accurate predictions online, i.e., *while* an application is still executing. Specifically, the technique yields an average accuracy of 84.0% when it is provided metrics pertaining to the first 30% of an application's complete execution (**RQ 2-5**).

Diaspore achieves an average accuracy of 85.6% over all the scenarios I consider. The technique's ability to provide accurate predictions with limited training data and offer online

predictions can allow Spark users to detect interference quickly and come up with effective mitigation strategies based on the identified root cause.

1.4 Publications

My research has been described in the following papers:

1. **Sarah Shah**, Yasaman Amannejad, Diwakar Krishnamurthy: "Diaspore: Diagnosing Performance Interference in Apache Spark", pages: 11 (in preparation).

2. **Sarah Shah**, Yasaman Amannejad, Diwakar Krishnamurthy, Mea Wang, "PERIDOT: Modeling Execution Time of Spark Applications", in Big Data Research, pages: 19, Elsevier, (submitted for review).

3. **Sarah Shah**, Yasaman Amannejad, Diwakar Krishnamurthy, Mea Wang, "Quick Execution Time Predictions for Spark Applications", in Proceedings of the 15^{th} International Conference on Network and Service Management (CNSM 2019), pages: 9, IEEE, Halifax, Canada, 2019, Acceptance Rate: 15.9%.

4. Yasaman Amannejad, **Sarah Shah**, Diwakar Krishnamurthy, Mea Wang, "Fast and Lightweight Execution Time Predictions for Spark Applications", 9^{th} International Conference on Cloud Computing (CLOUD 2019), pages: 3, IEEE, Milan, Italy, 2019.

1.5 Thesis Organization

The thesis is organized as follows. Chapter 2 offers background information on Apache Spark and the tools and benchmarks I use in the later chapters. Chapter 3 describes the

existing literature for modeling execution time of Spark applications and for identifying interference during the execution of Spark applications. Chapter 4 explains the details of Peridot, my proposed approach for predicting application execution time, and provides the evaluation results. Chapter 5 describes and evaluates Diaspore. Finally, Chapter 6 concludes the thesis by summarizing my two contributions, discussing the limitations of my research, and outlining future research directions.

Chapter 2

Background

In this chapter, I provide the background information on Apache Spark and the applications used in this thesis. Section 2.1 introduces the terminologies used in Apache Spark and describes the structure of a Spark application. Sections 2.2 and 2.3 describe the Spark applications used to evaluate my work and the tools used for injecting memory and processor interference, respectively. In Section 2.4, I provide an overview of the AutoML tool I have used in my research. Section 2.5 summarizes this chapter.

2.1 Apache Spark

Apache Spark is a big data analytics engine that can be used to support in-memory, distributed data processing. A Spark cluster typically has a single *driver* node interacting with several *worker* nodes. Each worker node executes operations on a specific portion of a large dataset. In this section, I explain the typical structure of a Spark application. I also describe Spark's data partitioning behaviour, which is leveraged in Peridot. I also discuss performance metrics that I have used in my experiments.

2.1.1 Spark Application

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. Figure 2.1 is a screenshot from the Spark UI that shows a small part of the Spark DAG for the Linear Regression (LR) application from the Spark standard examples. To see all the stages and jobs of this application, I have provided a simplified visualization of the full DAG in Figure 2.2 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. Figure 2.3 shows a small part of the Spark DAG for a Spark SQL query, Query 52 ($Q52$), from the TPC Decision Support (TPC-DS) [21] benchmark suite. TPC-DS is introduced in Section 2.2.3 of this chapter. From this excerpt, one can observe multiple stages within a job. Some stages run in parallel while others run in sequence. To aid better visualization of the dependencies, Figure 2.4 provides my simplified visualization of the DAG with parallel stages. Regardless of the size of input data or the resource allocation setting, the overall DAG structure remains the same for any given application.

Next, I focus on behaviour within each stage. To facilitate parallel processing, Spark

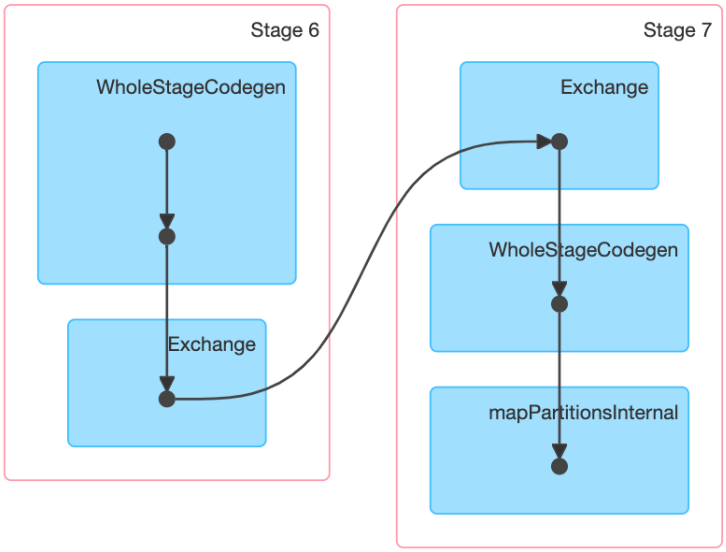

Figure 2.1: Example of a simple DAG

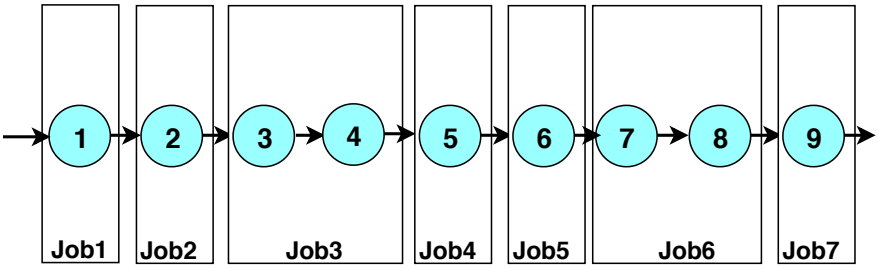

Figure 2.2: Simplified DAG structure of LR

splits the input data of each stage 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 my 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

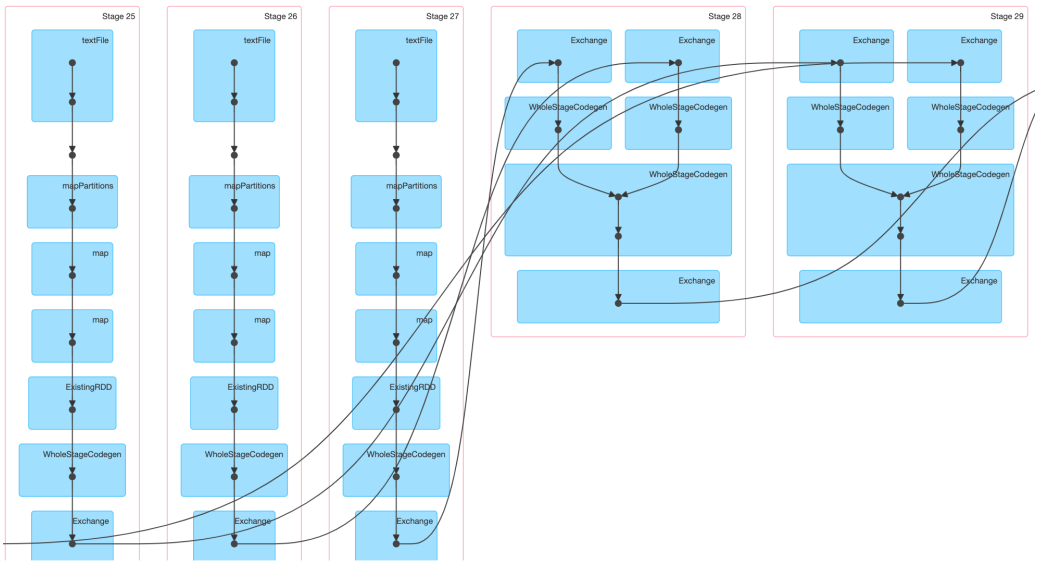

Figure 2.3: Example of a DAG with parallel stages

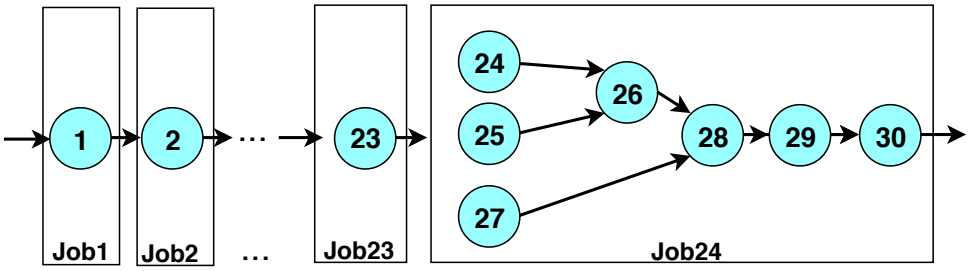

Figure 2.4: Simplified DAG structure for Query 52

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. I refer to the pattern of execution of tasks within a stage as a task *wave*. Wave time is defined as the time taken for one wave of execution. Figure 2.5 derived from Spark's visualization of actual stage executions shows two examples of tasks running in waves. In the example shown on top, 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

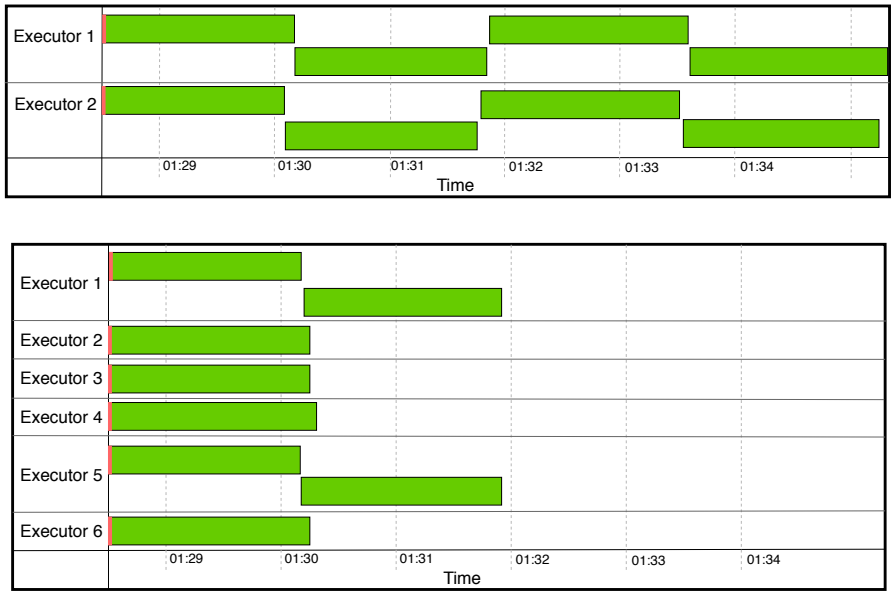

Figure 2.5: Examples of task waves

utilize the available parallelism. In the example shown at the bottom, 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. I 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. Figure 2.6 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 Figure 2.5.

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.

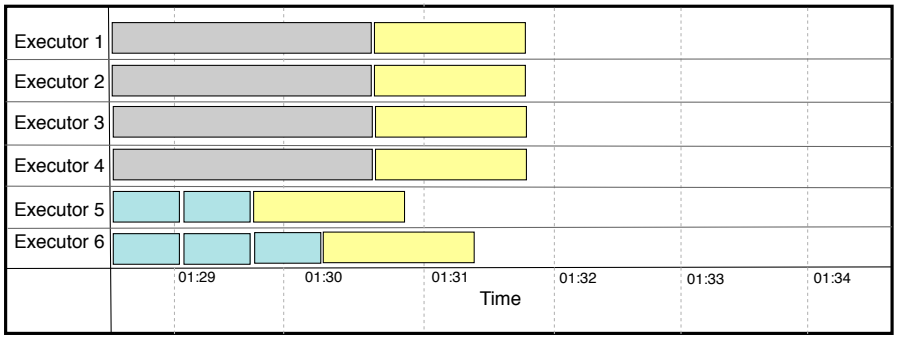

Figure 2.6: Examples of task waves in parallel stages

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. In Chapter 4, I exploit this observation to develop an analytic technique that captures the execution time of a Spark application.

2.1.2 Spark Application Metrics

The tasks within a stage are characterized by several metrics that can be used to monitor the stage they belong to and eventually the application as a whole. These metrics can be extracted from the executors running the tasks. Spark's official documentation [2] defines these metrics in detail. I refer to these metrics as *application metrics*. I will use these metrics in Chapter 5 to build my interference detection technique for Spark applications. They include:

- **Executor Run Time:** This is a metric that signifies the time spent executing the operations related to a task, including the time needed to shuffle the data,

i.e., transfer the data partitions for the task between different executors on worker nodes.

- **Garbage Collection (GC) Time:** This metric quantifies the amount of time taken by the associated Java Virtual Machine for garbage collection during the task's execution. GC Time ensures that any unnecessary data is removed from the memory heap so that there is enough memory for new RDDs. Since Spark runs in-memory operations and relies heavily on cached RDDs, GC Time becomes an important factor influencing performance. Significant GC times have previously been identified to cause performance problems or lead to complete task failures in Spark [22, 23].

- **Task Deserialization Time:** This represents the time spent by the executor deserializing the serialized Java objects dispatched by the main Spark driver program. This is a computationally intensive step that can influence application performance.

- **Scheduler Delay:** Scheduler delay is the sum of the time taken by the Spark scheduler to assign a task to an executor and the time taken to transmit the task's result from the executor to the scheduler. Scheduler delay can increase if there are a large number of tasks and if a task has preferences on the nodes on which it wants to execute for reasons such as data locality.

- **Task Duration:** This characterizes the overall time taken by a task from its initialization to end, including all operations and consequent delays associated with the task.

- **Max Task Duration:** This metric represents the maximum of all the task durations within a stage. Tasks within a stage can have very different execution times due to factors such as heterogeneity of cluster nodes, imbalanced data

sizes across executors, and varying magnitude of interference at the cluster nodes participating in the computation.

2.1.3 System Metrics

For studying interference, I also gather metrics that characterize the state of the node resources. I refer to them as *system metrics*. Since in this work, I study processor and memory interference specifically, I take note of the following system metrics:

- **Processor utilization:** This is the percentage of total computing capacity of the central processing unit being used at a given instance of time.
- **Memory Utilization:** This metric represents the percentage of system memory being used at a time.
- **Memory Commit:** This is the total amount of memory allocated for running the processes within a system, regardless of it being utilized by those processes.

2.2 Spark Application Benchmarks

To compile a diverse set of applications to evaluate the techniques I propose in this thesis, I use applications from well-known benchmarks. Since extensive experimentation on each application was required to gather data for evaluating the models, I picked a subset of applications from each benchmark. Each application within a subset chosen from a benchmark is distinct from the other selected applications to diversify my application set. A description of these benchmarks and the applications I pick from these benchmarks is provided next.

2.2.1 BigDataBench

I pick applications from a well-known benchmark for Big Data platforms, BigDataBench [24]. Specifically, I use Python implementations of Word Count (Wc) and Sort (So). These applications represent memory and processor demanding applications with a small number of

stages and simple DAG structures. They have been part of the standard example suite that is released with Apache Spark and are heavily used in existing literature for both problems I address in this thesis.

2.2.2 MLlib

MLlib [25] is Spark's ML library that offers a variety of ML algorithms spanning but not limited to regression, classification, and clustering. I have chosen 3 algorithms from this library. These include Linear Regression (Lr), Naive Bayes (Nb), and Correlation Analysis (Co). All of these applications generate multiple jobs, but they differ in their purpose, structure and resource utilization. They are further elaborated in Chapter 4.

2.2.3 TPC-DS Suite

TPC-DS [21] is a benchmark that offers SQL queries for modeling performance of decision support systems. It contains 100 queries that operate on 25 tables which can be scaled from 1 GB to 10 TB. I pick 7 queries to include in my application set. These include Queries 9, 15, 26, 52, 64, 70 and 78 ($Q9$, $Q15$, $Q26$, $Q52$, $Q64$, $Q70$, and $Q78$). These queries include a large variety of application structures and hence bring in various resource demands and functionalities in my application set. One of the main reasons for including the queries from TPC-DS benchmark is that the queries have parallel stages and jobs which was not captured by the previously chosen benchmark applications. This helps in building and testing the models more comprehensively.

I further expand my application set by adding a Matrix Multiplication (Mm) application that receives an $m \times n$ ($m >> n$) matrix, and multiplies the transpose of this matrix with itself to output an $n \times n$ matrix [26]. Mm and the other applications listed comprise a comprehensive set of 13 diverse applications that capture various application functions, DAGs, and resource demands. I use the same set of applications for validating both Peridot and Diaspore, in Chapters 4 and 5.

2.3 Tools Used for Interference Injection

To build a dataset for my interference detection model, I use microbenchmarks to initiate contention for memory and processor resources in a Spark cluster. Details of these microbenchmarks are provided next.

2.3.1 RAMSpeed

RAMSpeed is a tool to benchmark memory and cache performance in amd64, i386, and alpha based systems. *RAMSpeed* offers various functionalities like INTmem and FLOATmem that simulate computing through four different functions: Copy, Add, Scale, and Triad. Copy operation pastes data from one memory location to another. Add operation adds data from two different memory locations and writes it on another location. Scale operation reads data from one location and writes it on the destination location after scaling it by an integer. Triad operation joins the operations of Add and Scale and writes the result in the new location. I use INTmem to induce memory interference in Spark platform. I vary the number of instances of *RAMSpeed* running in parallel with my applications to vary the intensity of the interference. This helps collect data on interfered applications which is used to build and test my interference detection technique in Chapter 5.

2.3.2 MemArray

MemArray is a custom application designed to store a huge array within the cluster memory, thus limiting resources for other applications on that cluster. It is used as a source of memory contention for Spark applications used in preparing the dataset in Chapter 5.

2.3.3 Pi Calculator (P_i)

P_i is another custom application I use to calculate the value of π number up to a specified decimal place. Since, this is a mainly computation-intensive application, I use P_i in Chapter 5

for causing processor-based interference in Spark applications.

2.3.4 Prime Number Calculator (Prime)

Prime is an application for calculating prime numbers within a given range. I use *Prime* as a source of processor interference in the Spark applications used for building and testing Diaspore in Chapter 5.

2.4 AutoML

Many ML tasks require extensive manual effort for selecting a model and fine-tuning the model parameters by a trial-and-error process [27]. A common ML task starts with data preparation. In this step, experts are required to do different tasks including data cleaning, data exploration, or data augmentation. Then, data transformation techniques are applied to build new features or feature scaling techniques are used to standardize or normalize the existing data features. When data is ready, experts need to select an ML algorithm and train it. This usually involves training many different models with a subset of data called *training data* and then evaluating the trained models with an unseen subset of data, called *validation data*. A group of top performing models are then selected for further tuning of certain adjustable parameters i.e., hyperparameters. The hyperparameters of these models are tuned using trial-and-error randomized search or grid search processes. Finally, the best performing model is selected. At this time, a completely new subset of data called *test data* that has not been involved in any of the previous steps including data exploration, model training or hyperparameter tuning is used for testing and reporting the performance metrics of the final model. This whole process often requires substantial resources and time.

AutoML automates the manual effort in the ML model development. It has 4 main steps: Data Preparation, Feature Engineering, Model Generation, and Evaluation [19]. Using AutoML, many different variations of data features and models can be evaluated in parallel to

find the best performing model. In particular, AutoML typically exploits the computational capacity offered by cloud providers. Currently, many cloud providers, such as Amazon and Google, offer AutoML service for their users. I use AutoML [28] from Google Cloud Platform (GCP) for building my interference detection model in Chapter 5.

Google's AutoML suite offers a range of services such as AutoML Video Intelligence for classification and object tracking in videos, AutoML Natural Language for document processing and analysis, AutoML Vision for image processing, and AutoML Tables for building, testing and running large scale ML algorithms on structured data. I use AutoML Tables for generating and then deploying my model to expedite the model selection and tuning processes.

2.5 Summary

This chapter offers background information on Apache Spark and the benchmark applications used in my work. I describe the Spark and system metrics that help build the two techniques I propose for predicting execution times and diagnosing interference. Finally, I provide background on AutoML and how it is employed in my research to simplify the ML modeling process.

Chapter 3

Related Work

This chapter reviews the existing literature relevant to execution time prediction and interference diagnosis for Apache Spark applications. Specifically, it highlights research gaps that I address in my thesis. Section 3.1 discusses analytical and ML approaches developed to predict application execution times and to allocate resources for Spark applications to meet their execution time targets. In Section 3.2, I provide an overview of the interference detection techniques that have been proposed for Spark platforms. Section 3.3 summarizes this chapter.

3.1 Execution Time Prediction

There have been a number of recent efforts on modeling and optimizing the performance of Spark applications [4–9, 29–32]. Singhal and Singh [33] 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 with respect to input data and cores for execution. Petridis et al. [4] 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 these 12 configurable parameters for any Spark application's performance, in order to reach the best application time speedup. Wang et al. [5] proposed an ML 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 b effective. Obtaining such data might not always be feasible. For example, Jyothi et al. [17] analyzed workloads on production clusters and showed that about 40% of applications were non-recurrent, i.e., did not have historical execution data.

Some studies focused exclusively on the impact of data size and the amount of resources assigned to a Spark application. Gibilisco et al. [6] 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 [29] 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. [7] 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. [8] 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 [34] 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. [9] 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 [35] 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. Maroulis et al. [36] exploit similarities between different Spark applications to reduce the amount of training data needed for an ML based prediction. They validate their technique using a set of Spark applications with simple dependencies. Chao et al. [31] propose a gray-box approach which uses hardware configurations and resource allocation information to train the black-box ML model. Their technique involves training separate regression models for each stage from different configurations of an application and then using them to estimate overall application time. Similarly, Popescu et al. [30] also suggested modeling each phase of an application separately through a combination of analytic and hybrid approaches and then using these models to predict execution time.

Sidhanta et al. [3] 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. The profiling parameters require curve fitting over several experiments from the same application type and the cost optimization problem uses Interior Point Algorithm. Lee et al. [37] extended OptEx by including additional parameters to incorporate the probability of failure for an application and used this model to drive resource allocation. Since OptEx relies on curve fitting, it requires several reference executions, as opposed to my approach that need only two reference executions. Baresi et al. [38] proposed a containerized modification of Spark that exploits a black box control theoretic model to realize per-stage deadlines at run-time. Unlike my approach, their approach relies on modifying the default Spark system, which many not be feasible in production environments.

The extensive training evident in the literature summarized above motivates the need for modeling techniques that are less time and resource intensive. In comparison to the ML and curve fitting approaches discussed previously, Peridot takes a lighter weight approach to achieve comparable accuracies. Furthermore, in contrast to existing work, my work considers a much wider set of of applications including those with complex internal dependencies, e.g., TPC-DS queries, whose execution times are harder to predict. Peridot exploits knowledge of Spark internals such as the internal stage dependencies of applications, Spark's partitioning mechanisms, and the impact of task waves. I demonstrate in Chapter 4 that considering such information allows for the extraction of models that can provide accurate predictions with very little training data. I also show how such a technique could be exploited to predict the resources needed by an application to meet a specified deadline.

3.2 Performance Interference Diagnosis

As summarized above, there have been several efforts on modeling the performance of Spark applications. in the absence of interference. These approaches use historical executions of an application to derive a model that can predict the application's execution time under various configurations, e.g., different input data sizes and core allocations. Models derived from these techniques can be used to identify if the execution time of an application is abnormally high for a given configuration. However, the models are not intended to provide any information on the root cause of abnormal executions.

Ren et al. [10] proposed a technique to identify cluster nodes and tasks behaving abnormally. Their approach uses both task and stage performance metrics in conjunction with a logistic regression classifier to identify nodes causing the high execution time. In contrast to their work, my work focuses on interference and a finer-grained localization of root cause to specific node resources.

Similar to my work, several studies have focused explicitly on the impact of interference

on Spark applications. Some of them estimate the execution time in the presence of interference [11] while others provide insight into the root cause of the interference [12–15]. There have also been some efforts towards providing mitigation solutions based on analysing the root cause [16]. I discuss a selection of these studies in detail next.

Wang et al. [11] profile individual Spark applications and use that information to predict execution time slowdowns when multiple Spark applications are co-located on cluster nodes. The authors validate their execution time predictions on four Spark applications namely, Word Count, Page Rank, Logistic Regression, and K-Means.

Lu et al. [12] proposed a statistical anomaly detection technique that isolates problematic tasks in stages, locates the nodes running them, and identifies the root causes in the nodes behind the slowdowns. The anomalies are detected on a per task basis, by comparing the tasks to the average and standard deviation of task time in each stage. However, since the statistical approach does not provide good precision on classifying the root cause, the authors extend their work in a follow up paper [13]. In the extension, the authors develop a system that analyzes Spark application execution logs to identify resource bottlenecks, e.g., processor, memory, disk, and network, causing abnormally high execution times. The technique leverages a General Regression Neural Network model to predict the type of bottleneck. In contrast to my work, their work has been validated only on 3 applications. Furthermore, the technique proposed is designed for offline use, i.e., after an application has finished executing. My approach supports an online mode where the root cause identification can happen when an application's execution is still in progress.

Ding et al. [14] proposed a technique that can identify tasks negatively impacted by interference and nodes with significant interference at run-time. They rely on task metrics collected in real time and hence they can offer online predictions without waiting for all tasks in a stage to finish. Task metrics and system metrics are used with an Adaboost classifier to offer predictions. The technique is validated on three applications namely, Word Count,

Page Rank, and Naive Bayes. I validate my approach on a wider selection of applications and I also focus on finer-grained, node resource-level diagnosis of root cause.

Alnafessah et al. [15] proposed a neural network based approach to identify the root cause of interference. The work uses task-level metrics to support online prediction. Similar to my work, their work can localize root cause to individual node resources. However, the proposed model has been validated only on two applications namely, K-Means and a Spark SQL query. Furthermore, the model has to be trained separately for each of these applications and the training set includes both small scale and large scale executions. The authors use their final dataset in a followup paper [39] that aims to distinguish between processor, cache thrashing, and context switching related interference scenarios. Using the Word Count application, the paper shows good performance for detecting cache thrashing and comparatively lesser performance for the other two interference scenarios.

Zhao et al. [16] designed an interference-aware scheduling technique for Spark applications. Their approach estimates the processing capabilities of different nodes involved in a Spark computation by considering the intensity of interference in those nodes. The scheduling policy then ensures that more tasks are scheduled on nodes with less interference. The study only considers two applications that both contain only Map and Reduce stages. I consider a wider selection of applications with more complex operations in my work. Furthermore, I also identify root cause of execution time degradation. This can facilitate customization of mitigation solutions based on the root cause.

Table 3.1 summarizes the discussions in this section. From the discussions, existing work has not focused on broadening the validation of the proposed ML models to a wider range of Spark applications. Furthermore, most of the studies reviewed in this section develop a dedicated ML model for every application of interest and the training data for each application model needs to encompass different data sizes and core allocations. This level of intensive training effort needed to build models may not be feasible in practice.

Finally, existing studies have not focused on studying the generalizability of ML models to data sizes, core allocations, and applications not included in the training data. This thesis aims to address these gaps.

Table 3.1: Summary of interference detection techniques for Apache Spark

Ref.	**Diagnosis Level**	**Feature Level**	**No. of Apps**	**Generalizability**	**Online**	**Max Data**
[10]	node resource	Stage	3	Not explored	Yes	NA
[11]	node	Stage	4	Larger scale data	No	20 GB
[12]	node resource	Task	3	Not explored	No	NA
[13]	node resource	Task	5	Not explored	No	80 GB
[14]	node	Task	3	Not explored	No	300 GB
[15]	node resource	Tasks	2	Not explored	No	64 GB
[16]	node	Task	2	Not explored	Yes	12 GB
Diaspore	node resource	Stage	13	Larger scale, unseen apps	Yes	100 GB

3.3 Summary

Reviewing existing work on execution time modeling, it can be observed that many of these approaches rely on discerning the performance behaviour of a Spark application by analyzing a large number of historical executions of the application under different configurations. Gathering this data can be a long and expensive process. Therefore, Peridot offers a quick and lightweight alternative that will later be shown to offer accurate predictions for a wide range of applications. Similarly, analyzing the techniques proposed to diagnose interference shows that there is a gap for a technique that can offer predictions with limited training data, generalize to scenarios not captured in the training data, and be effective in

online settings. My research addresses these gaps with Diaspore.

Chapter 4

Execution Time Prediction for Spark Applications

In this chapter, I introduce Peridot, an analytic approach for predicting the execution time of Apache Spark applications. In Section 4.1, I explain the need for a quick tool for execution time prediction of Spark applications. Section 4.2 explains how Peridot captures the structure of each application, gathers information about the applications data and core parameters in a very short time and then uses this information to make predictions for different data or core sizes that the application may run on. In order to evaluate the effectiveness of Peridot on a wide range of application types and application settings, I have designed the experiment process in Section 4.3. The results are explained in Section 4.4. Section 4.5 elaborates a use-case of Peridot as a conservative resource allocator. This chapter is summarized in Section 4.6.

4.1 Motivation for Lightweight Predictions

As described in detail in Chapter 1, Apache Spark is 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.

There have been several research efforts in trying to model the execution time of Spark applications, which have been discussed in detail in Chapter 3. However, these techniques are generally very time consuming and tedious to train because of the large amount and variety

of historical data they require for making accurate predictions. The availability of such historical data for each application can be problematic when the time and resources required to generate this data are unavailable. To address these concerns, I 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 runs an application on a small fixed number of cores with two different-sized, small subsets of its input data, and extracts the internal dependencies in the application as well as the impact of data partitioning and data size on execution times in these runs to deduce the execution times of the application under other resource allocation settings and input data sizes. This helps in reducing the time and resources required to construct application models significantly, thereby facilitating quick insights into the execution time behaviour of applications.

4.2 Execution Time Modeling with Peridot

Peridot is a lightweight, analytic algorithm for execution time estimation that bypasses exhaustive prior data of an application's executions in different settings to derive a model. Given an application, its input data, and the execution time target, Peridot launches just two different significantly smaller executions of the application on minute subsets of the input data to figure out the amount of resources that can achieve the execution time target, so as to not consume a large 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, and are referred to as *reference executions*. Using these two executions, Peridot follows a three step process to make predictions for the execution time of the application with other input sizes and cluster cores.

The **first** step in this process is estimating the dependencies among stages in the application. This is followed by looking into the execution logs, to determine the number of data partitions processed in each stage and whether the number of these partitions changes with

the input size (Section 4.2.1). The **second** step is to calculate the number of waves in each stage or a group of parallel stages and then determine the average wave time (Section 4.2.2). The mean wave times for each stage and other stage characteristics extracted from the application logs are used in the **third** step of this method to estimate the execution time of the application with different input sizes and executor core numbers (Section 4.2.3).

4.2.1 Application Dependency Identification

This is the first step in implementing Peridot, where the Spark application is run twice with two small input data sizes on a small number of executor cores. These two runs are called *reference executions*. The execution logs of both reference executions are collected to obtain detailed information such as stage id, tasks to be processed in each stage, start time of every stage, the time at which the first task of a stage starts executing, and the time at which a stage finishes. From these reference logs, the dependencies between the stages are extracted and the application's partitioning behaviour is characterized based on changes with size. It is important to note that the dependency between stages within an application is defined by the operations that the application is meant to execute and remains unchanged even when the input data or the executor cores assigned to the application are changed.

Next, groups of stages are identified and the application is represented as a sequence of these groups. A *stage group* is defined as a set of stages scheduled at the same time. A stage group may contain only one stage, thus implying that the stage is scheduled sequentially with respect to other stages. Stage groups containing multiple stages indicate parallel stages i.e., these stages are executing at the same time. In Chapter 2, I showed a simplified DAG structure for $Q52$ in Figure 2.4. As it can be seen from this figure, there is no other stage scheduled to run in parallel with stage 1, which is why it will have its own stage group with only one stage. Since no other stage is scheduled simultaneously, this stage will run in sequential order. Stages 24, 25, and 27 however are parallel stages since they are scheduled at the same time and hence they are all grouped together in one stage group.

Because of simultaneous execution, parallel stages are bound to compete for the processing resources assigned to the application. Consequently, a wave at any given instant between the start and finish times of these stages contains tasks from different stages. This makes the evaluation of multi-stage groups much more complex than stage groups that contain only a single stage. Regardless of whether a stage group is sequential or parallel, the execution time S_i of the stage group i is defined 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. This metric captures delays due to tasks waiting for executor cores in addition to merely capturing the computation time of tasks. The identification of stage groups based on stage scheduling time yields a sequence of stage groups estimating the dependency structure of the application.

I take 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 Figure 2.4 from Chapter 2, Spark's DAG scheduler will schedule stages 24, 25, and 27 at the same time. Accordingly, my 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 re-creation and capture of the dependencies would require a detailed simulation of the DAG's scheduling and execution. In contrast, my simplified approach allows me to analytically estimate execution times without resorting to simulation. The ability to offer quick predictions makes my approach well-suited for use within job schedulers at runtime for "right sizing" an application. Furthermore, the analytic nature of my approach allows it to be used within job planning optimization frameworks. I evaluate the effectiveness of my simplification in Section 4.4.

After extracting the sequence of stage groups from each reference execution log, I compare the corresponding groups in the references to characterize partitioning behaviour. Specifi-

cally, I 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 my 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 based on whether there is a change in the number of partitions processed by them.

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, I extract the number of partitions in each stage group under each of the two reference executions. Peridot also extracts the stage group execution times and the end to end execution times from both references.

4.2.2 Model Parameter Extraction

Based on the outputs from the previous step, Peridot first calculates the *mean wave time* parameter for each variable stage group. As shown later in Section 4.4, 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 ML dataset, remain unchanged. Since Peridot runs the reference executions on two small subsets of the original input data that the application would run on, the structural properties do not change between the reference and the actual executions of the application. Consequently, knowledge of the mean wave time allows offering predictions for any executor cores and data size as outlined later in Section 4.2.3.

The number of waves N_i in stage group i can be estimated using Equation (4.1) 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 group S_i by the number of waves N_i, as shown in Equation (4.2). Peridot calculates W_i in this manner using each of the two references. The mean of these two values, $\hat{W}_i$, is then used as the mean wave time parameter for stage group i.

Parallel stage groups can exhibit variability in task execution times since tasks can belong to different stages within the stage group. Peridot's method of calculating wave times for such stage groups in essence averages the execution times of these heterogeneous tasks.

$$N_i = \lceil \frac{P_i}{E} \rceil \tag{4.1}$$

$$W_i = \frac{S_i}{N_i} \tag{4.2}$$

The mean number of partitions, $\hat{P}_i^r$, in each stage group is also recorded as a parameter for that stage group. This is calculated as the sum of the corresponding $\hat{P}_i^r$ values in both references divided by the mean of the input data sizes of the two references $\hat{D}^r$. As shown later in Equation (4.3), $\hat{D}^r$ is used to scale the number of partitions in each stage group while predicting for other input data sizes.

Finally, Peridot estimates 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. Peridot sums the variable stage group execution times, i.e., the S_i values, and subtracts this sum from T to obtain F. This process is carried out on both reference executions to obtain a mean fixed execution time $\hat{F}$.

4.2.3 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, Peridot estimates the new number of partitions, P_i^{new}, that result under these settings using Equation (4.3). This equation assumes that the number of partitions scales linearly with data size, which is the

behaviour of Spark's default partitioning algorithm. Next, the number of waves, N_i^{new}, is estimated using Equation (4.4). Equation (4.5) is used to estimate the execution time S_i^{new} of the stage group using the mean wave time parameter $\hat{W}_i$ calculated 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 to obtain the execution time prediction T^{new}.

$$P_i^{new} = \frac{D^{new}}{\hat{D}^r} \times \hat{P}_i^r \tag{4.3}$$

$$N_i^{new} = \lceil \frac{P_i^{new}}{E^{new}} \rceil \tag{4.4}$$

$$S_i^{new} = \hat{W}_i \times N_i^{new} \tag{4.5}$$

$$T^{new} = \sum_{i \in var} S_i^{new} + \hat{F} \tag{4.6}$$

4.2.4 Discussion

Peridot is compact since it captures both executor and data scaling using the number of waves as shown by Equations (4.3) and (4.4). 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 simplifications with regards to the dependency structure, the characterization of fixed and variable partition stages, and the wave behaviour of stage groups. The impact of such simplifications on accuracy is studied experimentally in Section 4.4.

As mentioned before, 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 4.5.

4.3 Experiment setup

4.3.1 Experiment Testbed

I have evaluated my approach on two clusters consisting of nodes from the ARC cluster of University of Calgary [40]. My 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 my technique can be generalized to other platforms, I 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. The test environment is equipped with Spark 2.2.0 and operates on a common network file system.

4.3.2 Experiment Factors

Applications - I evaluate my technique with 13 standard benchmarks encompassing text analytics, linear algebra, ML, 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 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

ML 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 my evaluation, I have also used a Matrix Multiplication (Mm) application [26]. 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 my test applications, I also use Spark SQL queries from the industry-standard TPC-DS benchmark suite [21]. I include these queries because of their more complex DAG structures compared to the rest of the applications. Specifically, I use Query9, Query15, Query26, Query52, Query64, Query70 and Query78. I 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 my knowledge, no previous works have explored applications with parallel jobs. $Q15$ is a reporting query that accesses only one table in the TPC-DS dataset. $Q26$ and $Q52$ are interactive queries that have been used by other researchers developing performance models [41]. $Q64$ 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 [42], hence providing a well-rounded testbed for Peridot.

Table 4.1 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 I have selected cover the three most frequently observed communication patterns in Spark [8] namely, Collect, Tree Aggregation, and Shuffle.

Input and resource configurations - I vary the input size from 1 GB to 100 GB. To

Table 4.1: Application characteristics

	# of Jobs	# of Stages	Parallel Stages	Collect	Tree	Shuffle
Wc	1	2	No	✓	–	✓
Mm	2	2	No	–	–	✓
So	2	2	No	–	–	✓
Co	33	53	No	–	✓	✓
Lr	7	9	No	–	✓	✓
Nb	13	15	No	✓	–	✓
$Q9$	41	56	Yes	✓	–	✓
$Q15$	26	34	Yes	✓	–	✓
$Q26$	25	35	Yes	✓	–	✓
$Q52$	25	31	Yes	✓	–	✓
$Q64$	26	75	Yes	✓	–	✓
$Q70$	25	38	Yes	✓	–	✓
$Q78$	25	43	Yes	✓	–	✓

process this data, I employ up to 40 executor cores distributed over 1 to 8 machines. Each executor in my experiments is configured with 50 GB of RAM, and 5 cores. Each node is configured with 1 executor, so my experiments range from 1 to 8 executors. I have studied other core assignments in Section 4.4.2 where I vary the number of cores assigned to each executor from 1 to 10. The choice of input sizes and executor core numbers allows me 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 Java Virtual Machine (JVM) process that coordinates the executors, and the operating system.

Other approaches - I 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.

I also compare Peridot with an existing ML based prediction approach from literature namely, Ernest [8]. As shown in Equation (4.7), 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*.

$$\begin{aligned} T = \theta_0 + \theta_1 \times (D \times \frac{1}{nodes}) \\ +\theta_2 \times log(nodes) \\ +\theta_3 \times nodes \end{aligned} \tag{4.7}$$

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

4.3.3 Experiment Process

For each experiment in the prediction process, I have an initial phase where two reference executions are carried out as described in Section 4.2 to extract the application dependencies and the model parameters. The reference executions use 5 cores. I use 1 GB and 2 GB data sizes in the reference experiments. To evaluate the prediction accuracy, each Spark application is run with all executor core settings and input sizes to collect the application execution times. I repeat each measurement three times and use the mean of the measured execution times. I 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, I start with an initial execution time target that will help meet a user specified deadline. I then gradually relax that execution time target up to 200% of the initial target. For each execution time target explored in this manner, I first search the minimum amount of required cores for each application based on Peridot's predictions. Then, I run the application with the selected amount of resources and compare the measured execution time of the application with its predetermined execution time target. I 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.

4.3.4 Experiment Metrics

To evaluate the accuracy of Peridot's predictions, I use the relative error of the predicted execution time values from the measured execution time values as shown in Equation 4.8.

$$Error = |\frac{T_{predicted} - T_{measured}}{T_{measured}}| \times 100 \tag{4.8}$$

Since I ran each application multiple times in any core-data setting, I am also interested in comparing the 90% Confidence Intervals (CIs) of actual execution time measurements with the predictions. I refer to the lower bound and upper bound of the measured CI as $LB_{measured}$ and $UB_{measured}$, respectively. I 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 Equation (4.9).

$$\begin{aligned} dist = Min(&|T_{predicted} - UB_{measured}|, \\ &|LB_{measured} - T_{predicted}|) \end{aligned} \tag{4.9}$$

Equation (4.9) 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.

I then define the confidence interval gap CIG using Equation (4.10). CIG is obtained by normalizing $dist$ by the measured application time $T_{measured}$.

$$CIG = |\frac{dist}{T_{measured}}| \times 100 \tag{4.10}$$

Finally, to compare the effectiveness of Peridot's predictions for selecting resources for an application, I compare the number of cores suggested by Peridot 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 of cores $T_{measured}$ exceeds the execution time target T_{max}, I calculate and report an overshoot metric os as shown in Equation (4.11).

$$os = \frac{T_{measured} - T_{max}}{T_{max}} \times 100 \tag{4.11}$$

4.4 Results

This section is organized as follows. I first provide an overview of the performance of the 13 applications I consider in Section 4.4.1. Then, I evaluate the validity of my prediction approach in Section 4.4.2 by running applications on different number of cores and observing the mean wave times. Sections 4.4.3 measures the accuracy of Peridot. Section 4.4.4 evaluates Peridot and compares its performance with the other two prediction techniques. Finally, Section 4.4.5 evaluates Peridot on multiple clusters. I have made the execution scripts and the logs of all experiments available online [18].

4.4.1 Application Performance Characteristics

I first show how the execution times of the 13 applications used here change with the number of executor cores. On my experiment testbed, the execution times vary by factors

of 3 to 25 while using the executor cores and size settings presented in Section 4.3.2. As shown in Figure 4.1, 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.

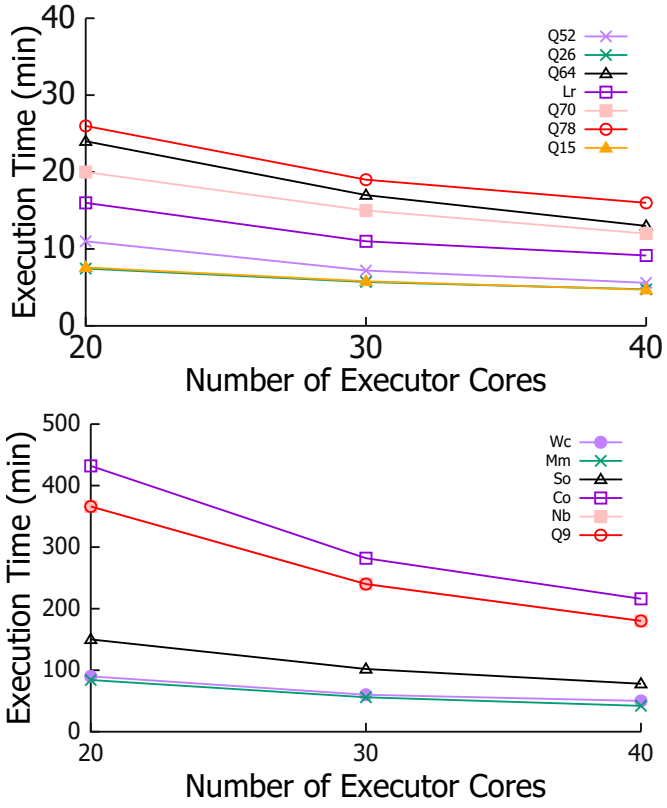

Figure 4.1: Execution time of applications

Figure 4.2 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 I've omitted 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.

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

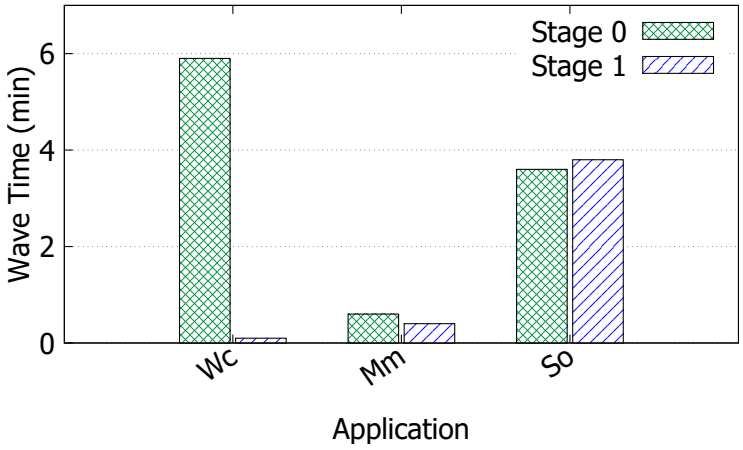

Figure 4.2: Wave time of the different stages of the applications

I next characterize an application with a parallel stage group. Figure 4.3 shows the mean wave times of various stages within one parallel stage group of $Q64$. From the figure, a single wave is comprised of different parallel stages and hence the individual stage task times can vary significantly. As described previously, Peridot reflects this behaviour by using all the tasks in the group to compute the mean wave time. This mean wave time is then used to make predictions for the parallel stage group in a different data-core setting.

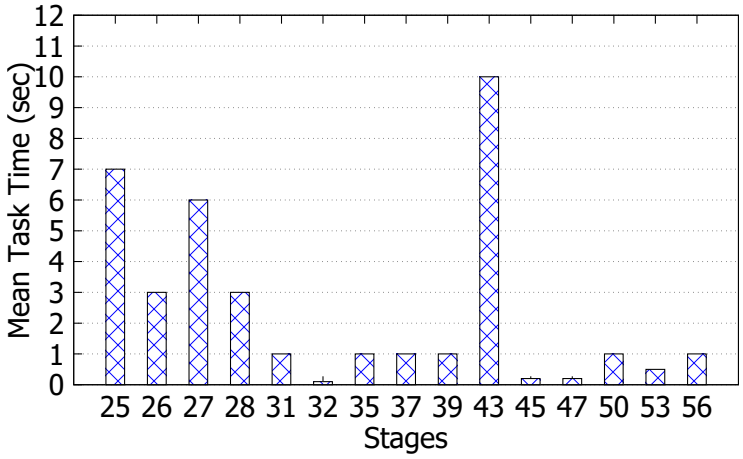

Figure 4.3: Variation in task execution times in parallel stages of Q64

Next, I 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 for a given stage of an application remains almost the same in any given scenario. 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). Figure 4.4 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 my modeling choice. I observe similar behaviour in other applications including those with parallel stages.

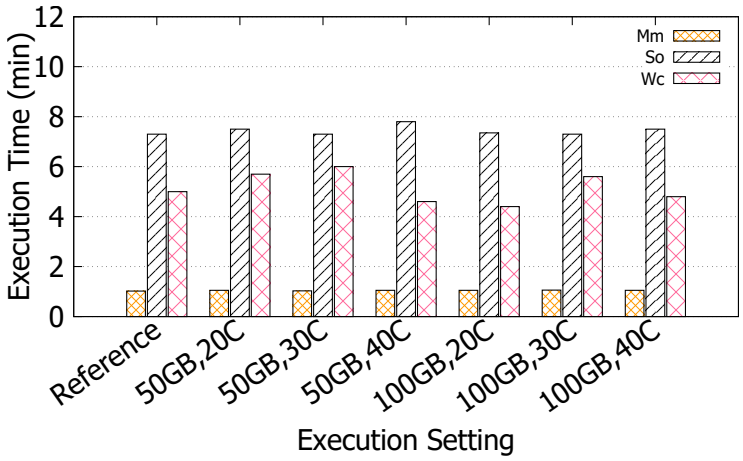

Figure 4.4: Wave time of the applications with different settings

Second, the reference execution uses data sampled from the target data and the sampled data has the same structural characteristics as the original data. I 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, Figure 4.5 shows that the average 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.

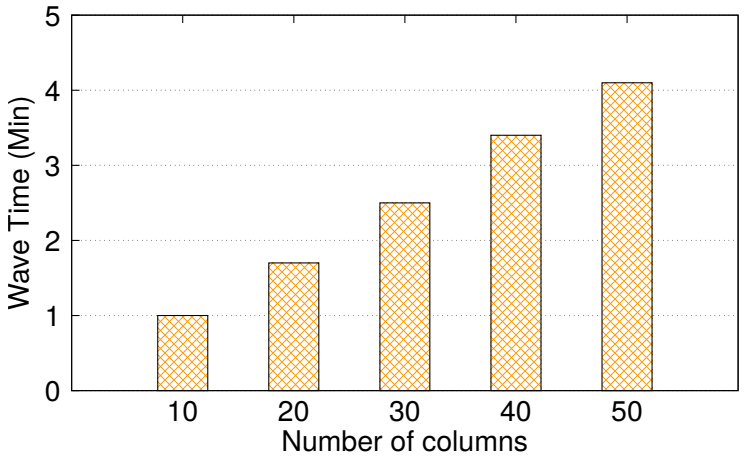

Figure 4.5: Effect of number of columns on wave time of Mm

Finally, Table 4.2 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 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 4.2: Stage types characterization

	Parallel Stage 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

4.4.2 Effect of Processing Core Assignment

Peridot uses the mean wave times measured from the reference executions to offer predictions. I 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. I collect the wave times of 4 different Spark applications under various cores assigned to a Spark executor on a single node. I study Mm, Lr, $Q70$, and $Q64$ to cover different application types and application complexities. Each application is run with 2 GB of input data and the number of cores is varied from 1 to 10. Figure 4.6, demonstrates the similarity in the wave times of the longest stage group of the applications under study under all core settings. This behaviour is observed 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.

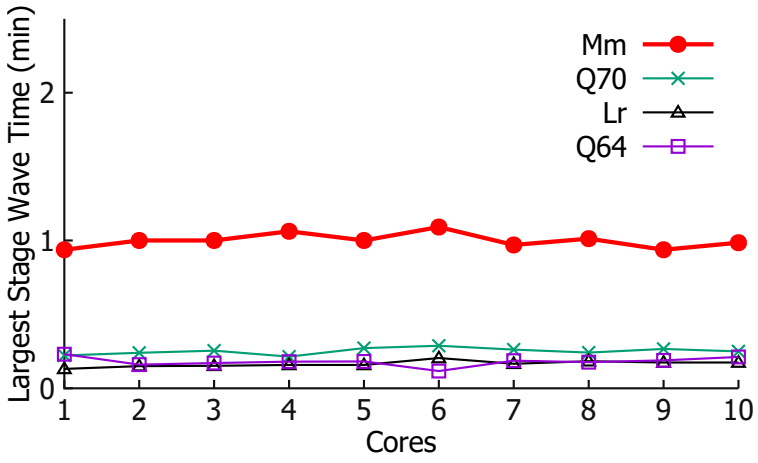

Figure 4.6: Wave times under different core assignments

4.4.3 Accuracy of Peridot

Figure 4.7 shows the mean prediction errors of Peridot for the 13 applications on my default cluster, *Breezy*. I 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% CI 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

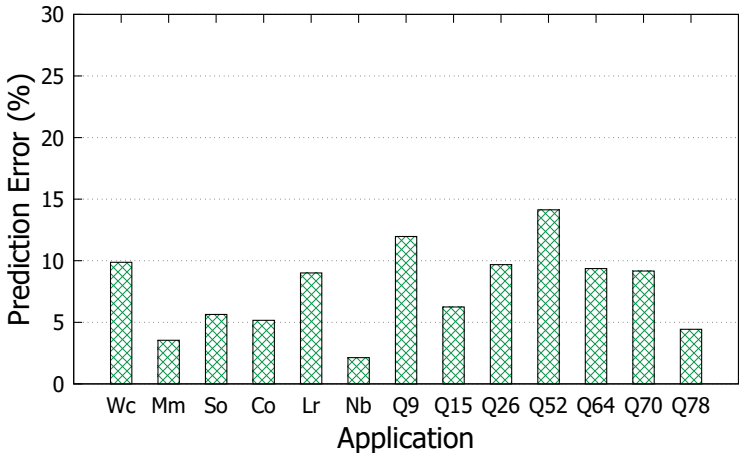

Figure 4.7: Prediction errors of Peridot on Breezy cluster

slightly higher mean error of 9.3%. In my 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 I will explore as future work.

Peridot provides quick predictions. For instance, exhaustive experimentation of all possible configurations I 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.

4.4.4 Comparison with Other Approaches

Figure 4.8 compares Peridot and IS. Peridot significantly outperforms IS offering predictions that are 10 times more accurate overall than IS. For the Wc, Mm, So, 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.

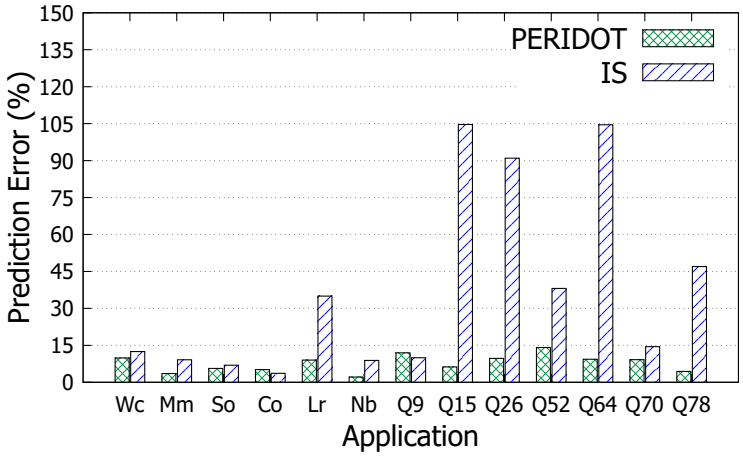

Figure 4.8: Prediction errors of Peridot and IS

Finally, I compare Peridot with the state-of-the-art ML 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 [8]. In this study, I 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 that 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.

4.4.5 Effect of Cluster Nodes

To explore how my modeling technique can be generalized, I evaluated Peridot on the *Bigmem* cluster [40]. Each node in this cluster has a very different resource setting than the *Breezy* nodes. I run the same set of experiments by varying the input data and core sizes and record the measured execution times. I then compare them against the predictions of Peridot. On *Bigmem*, I achieved the mean error and mean CIG of 10.8% and 9.7%, respectively over all 13 applications. The individual mean error for each application can be observed in Figure 4.9. Considering both clusters and all applications, Peridot achieves a mean error and mean CIG of 9.3% and 6.9%, respectively. These results validate the modeling choices described in Section 4.2.

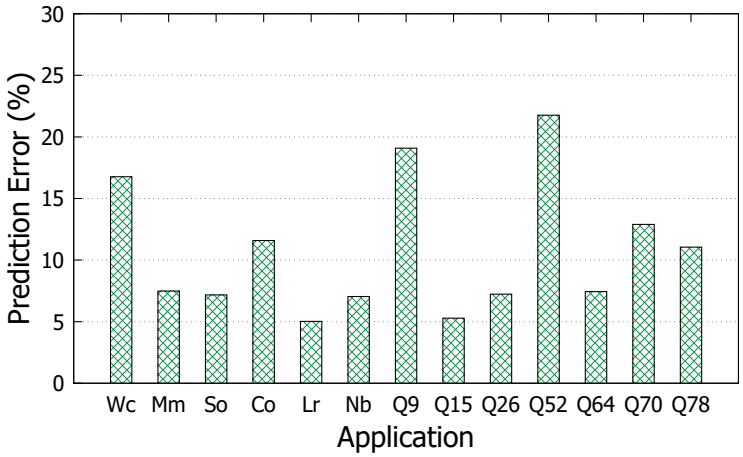

Figure 4.9: Prediction errors of Peridot on Bigmem cluster

4.5 Using Peridot for Resource Allocation

In this section, using an example, I 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, I 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, I start with an execution time target, and gradually relax that target. I assume my system to have a maximum of 50 execution cores that can be assigned to any application. To make the comparison easier, I 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, I conduct 10 additional experiments for each application, where I relax the execution time target of the application from 20% to 200% of the initial execution time target.

Figure 4.10 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. I 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, I 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 my experiment set. Considering all applications and scenarios I 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 an execution time target violation, I record the amount of the overshoot percentage *os*. Considering all 13 applications and the 10 time targets that I study for each application, the mean *os* is 8.3%. This is a reasonable value compared to the 43.1% resource conservation on average provided by Peridot.

Table 4.3 compares other static allocation strategies in terms of execution time target

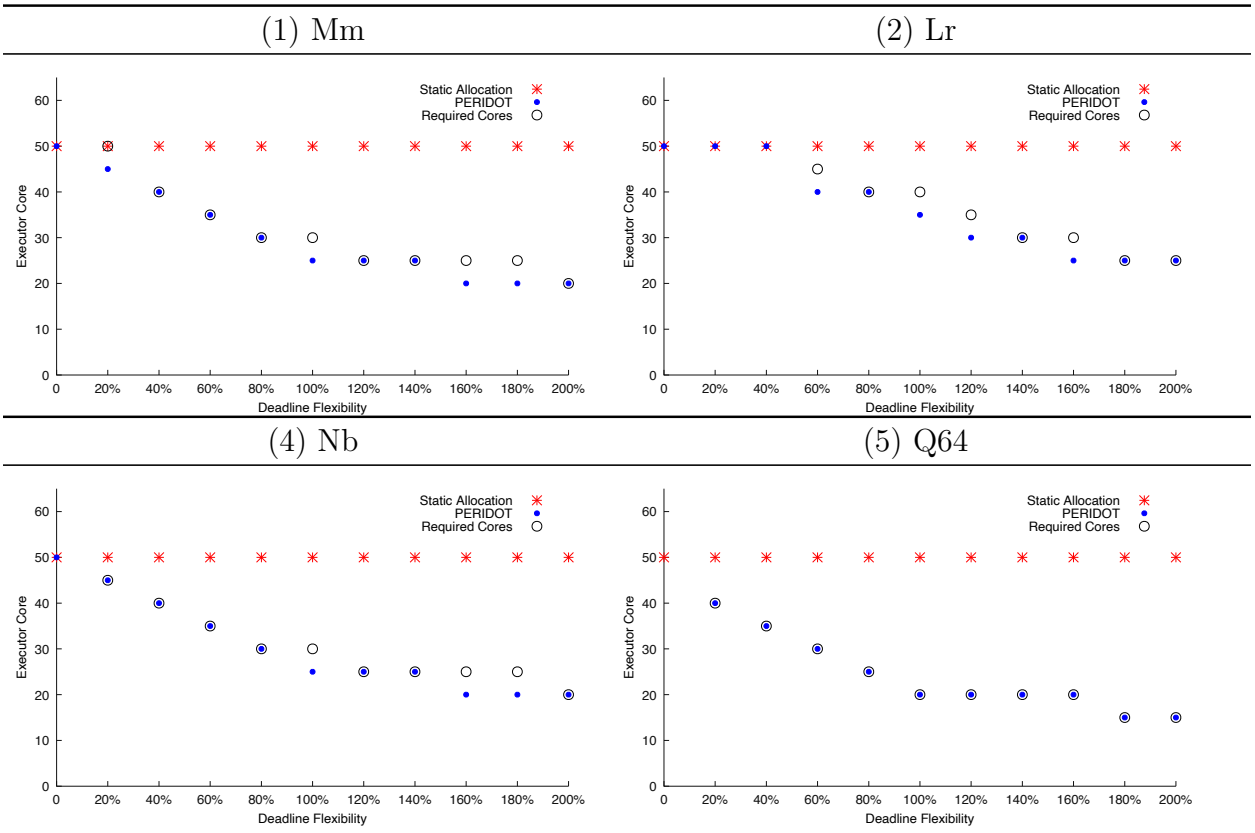

Figure 4.10: Executor cores allocated with the default approach, Peridot, and actual required cores

violations and resource conservation. As expected, assigning lower number of cores realizes resource savings but results in target time 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.

Table 4.3: Resource savings and 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%

4.6 Summary

The results in this chapter show that Peridot can accurately predict execution times for a wide range of application types. Peridot's fast and accurate execution time estimations can be used by Spark job schedulers to quickly configure any given Spark application submitted to a cluster for a desired execution time target. Moreover, if the expected execution time information is available for an application at the beginning of its run, it is easier to spot anomalous behavior if the application encounters any interference. I focus on interference detection and diagnosis in the next chapter.

Chapter 5

Interference Diagnosis in Apache Spark Applications

This chapter describes Diaspore, my proposed approach for interference diagnosis in Spark Applications. Section 5.1 revisits the motivation behind building an interference detection technique for Spark. Section 5.2 describes the dataset used for building Diaspore, the features incorporated in this dataset, the ML performed on the dataset, and the deployment of Diaspore for online predictions. Section 5.3 explains the data gathering and experimentation setting for building and evaluating the model. Section 5.4 details the results of all experiments and elaborates on the significance of making the modeling choices specific to Diaspore. Section 5.5 concludes this chapter.

5.1 Interference in Spark Clusters

As described in the previous chapters, resources are allocated to the applications running on Spark based on the prediction of application time that would result with those resources. However, due to a variety of reasons, the execution of an application may deviate from its usual behavior. If the machines are not faulty, then the reason behind unusual application executions is most likely interference. The other applications located on the same machines may compete for the system resources available to a Spark application, thus causing contention and slowing down the application's execution. Depending on the type of the application, the type of interference and the intensity of the interference, the application can experience serious delays which can be very frustrating to users and operators.

There have been a few efforts recently for detecting the interference in Spark applications which I have elaborated in Chapter 3. However, the existing models are often specific to an application or a certain type of applications. Moreover, the ML models have to be trained for

different scales of the application in order to successfully predict for different configurations. In this work, I propose Diaspore, an ML based technique for detecting interference in Spark applications that addresses these issues. The ensuing sections describe the data collection process, the choice of various metrics and their impact on the prediction process.

5.2 Methodology

Simple rule-based techniques may not be adequate for diagnosing the root cause of interference [13]. This motivates the Diaspore approach of exploring ML techniques. Section 5.2.1 provides an overview of Diaspore. Section 5.2.2 describes the data collection process and the features used in Diaspore's model. Section 5.2.3 describes the data processing step while the model selection and training process are presented in Section 5.2.4. Section 5.2.5 explains how Diaspore can be deployed to detect the root cause of performance interference in a Spark cluster.

5.2.1 Overview

Diaspore uses an ML model for identifying processor and memory interference in Spark clusters. To build the model, a set of controlled experiments are run in a sandbox environment within a cluster where the presence, type, and intensity of interference can be controlled. My previous work has suggested that small scale executions of a Spark application can be good predictors of larger scale runs [18]. Diaspore exploits this observation by only relying on small scale executions of a set of applications of interest under both types of interference with varying intensities. From these executions, application metrics are collected from the application execution logs. Furthermore, cluster nodes involved in the computation are profiled to obtain system metrics capturing processor and memory utilization. Diaspore processes the collected data and the processed data is used to select and train an ML model. The trained model is then used to diagnose any given execution of an application

from the set, including larger scale executions of the application. The classification identifies whether the application is impacted by interference. If so, the model also identifies the type of resource contention, i.e., processor or memory, driving the interference.

Consistent with the objectives described in Chapter 1, Diaspore limits the need for extensive training data. This is achieved by developing one global ML model for all applications in the set and by only relying on small scale executions to train the model. Due to these design choices, only modest cluster resources are needed for the training sandbox thereby making my technique practical. I show in Section 5.4 that the model obtained using this technique is able to generalize well to input data sizes, core allocations, and applications not seen in the training data.

5.2.2 Data Collection

To select and train a model, a set of controlled experiments are executed to collect three categories of executions. For each execution, two types of metrics are collected. For each type of metric, data is collected at two levels of granularity. I now describe these categories, types of metrics, and granularity of the metrics.

I consider a set of 13 well-known Spark applications to select, train and evaluate a model. Each application is executed when it is not experiencing any interference, and also when it is exposed to either processor or memory interference leading to three categories of executions. I focus on these two resources for interference detection because they are generally harder to distinguish from each other. These two resources are most commonly needed for the applications, and are limited in cluster settings. Moreover, several studies have previously excluded memory from their analysis. Another reason for this decision was that since I had limited time, I was able to generate an extensive set of experiments only for memory and processor related interference.

For selecting and training a model, only executions with 1 GB input on 1 cluster node and 2 GB input on 2 cluster nodes are considered. I refer to the 1 GB execution of each

application with no interference as its *reference* execution and use it for feature scaling of application metrics during the pre-processing step.

As discussed in Chapter 2, the application metrics collected are executor run time, GC time, task deserialization time, scheduler delay, task duration, and the maximum task duration. These metrics are collected for every stage in an application. Since this work focuses on processor and memory interference, I use the Linux *sar* utility to record processor utilization, memory utilization, and memory commit (percentage) at the cluster nodes every 2 seconds.

In my experimental evaluation, I compare the accuracy of the model when trained on only application metrics and when trained with both application *and* system metrics. In many systems, e.g., public cloud-based clusters, cluster users only have access to application metrics. To study the feasibility of users in such environments building models to diagnose interference, I explore models trained only with application metrics. I also train a model with both application and system metrics to represent the perspective of conventional cluster operators, who have access to both types of metrics.

Application and system metrics are collected at two granularity levels namely, *stage-level* and *task-level*. At both granularity levels the same metrics are collected, except for the maximum task duration, which is not relevant at the task level. To obtain application metrics at the stage level, the mean of each task metric is calculated over all the tasks within that stage. Additionally, the maximum of the observed task durations in that stage is recorded. When collecting system metrics for task-level or stage-level data, I use the mode values for the metrics during the time window of the task's or stage's execution. The mode utilization values represent the most frequent state of system resources observed while a task or stage is executing.

A number of studies reviewed in Chapter 3 use task level metrics to build models. This approach can be beneficial since it will yield more training data as each stage will typically

have many tasks. Furthermore, the use of task metrics can enable faster online interference prediction, i.e., predictions can be offered without the need to wait for the entire stage to complete. However, individual task execution times within the same stage can display considerable variability even when interference is not present. Section 5.4 studies the impact of this variability on the performance of the ML model.

The use of mean values for application metrics with stage-level granularity can smooth out normal task execution time variability that can exist even without interference. However, the stage-level approach needs to be adapted to offer online predictions because extracting and scaling the values of the metrics in the feature set of the model would require a stage to completely finish executing. Diaspore explores an approach to adapt a model trained using stage-level data to offer online predictions. Specifically, Diaspore employs a prediction threshold t that defines the fraction of tasks in a stage that need to complete before a prediction can be offered for that stage. While a setting of t= 100% offers offline prediction, lower values can be used to offer more agile predictions. With this scheme, Diaspore waits till the specified fraction of tasks have completed in a stage. Stage metrics are computed based on the completed tasks and those metrics are provided to the trained model to extract a prediction. I note that such partial stage data is used only during prediction. The model selection and training is performed with the complete stage data. I explore the impact of data granularity and the use of the prediction threshold t in Section 5.4.7.

Diaspore constructs data records from the metrics collected at different granularity levels from different categories of executions. Each record is represented by a vector containing the collected metrics and a label indicating the categories of execution, i.e., whether the application suffered from processor interference or memory interference, or executed normally. I create distinct datasets from the training executions to study the impact of the type of metrics and the granularity of the metrics.

5.2.3 Data Pre-Processing

Data records from the raw metrics are pre-processed before using them to select and train an ML model. Generally, ML requires data pre-processing as a preliminary step before model building. This could include data cleaning, feature scaling in the form of normalization or standardization, and feature engineering in the form of combining or transforming features. For pre-processing my data, I perform feature scaling, where I normalize the application metrics with respect to the reference values of those metrics.

As stated in Chapter 1, a key objective of this work is to explore the feasibility of one global technique for all applications to limit training effort. It follows from this requirement that the features I use in the technique should emphasize the degree of change in the values of metrics due to interference instead of their absolute values. Consequently, Diaspore performs feature scaling. Each metric in the feature vector pertaining to an application's execution is divided by the corresponding metric in the reference execution, i.e., execution with 1 GB and no interference, of the application. The scaled feature vector thus quantifies the degree of change in the metrics under the influence of performance interference with respect to the no interference reference.

While performing online predictions using partial stage data, the metrics collected over the incomplete stage are scaled with the corresponding metrics from the reference execution. For the task dataset, each application metric is scaled with respect to the mean value calculated for that metric in the respective stage from the reference execution.

I note that my feature engineering approach allows me to explore the generalizability of the model. Specifically, the feature vector does not explicitly encode the identity of the application, the data size, and core allocations. This design choice allows me to characterize in Section 5.4 the ability of Diaspore to offer predictions for scenarios not captured by the training data.

5.2.4 Model Selection and Training

To automate the selection of the best classification model for a given dataset and to also fine tune the hyperparameters of the selected model, Diaspore leverages the AutoML system that is part of the GCP. AutoML automates the entire ML process from feature extraction to model selection, hyperparameter tuning, and model evaluation. Since Diaspore explicitly performs feature engineering, AutoML is leveraged to mainly automate model selection and hyperparameter tuning. AutoML exploits the underlying cloud platform to launch a large number of parallel experiments. This allows the system to quickly search for the best model and hyperparameter settings.

5.2.5 Using the Model for Predictions

The model selected by AutoML is deployed for predictions as follows. When a new application execution is submitted to the cluster, Diaspore first searches historical executions performed previously in the sandbox to check if a reference execution exists for the application. If a reference exists, it is used to scale the metrics arising out of the submitted execution. The absence of a reference execution indicates that the execution pertains to a new application. In this case, Diaspore only requires the reference execution to be launched in the cluster sandbox once. As mentioned previously, the reference execution is quick and does not consume significant resources since it is performed at small scale. Metrics from the new reference execution are then used to scale the metrics collected from the submitted application execution. For both existing and new applications, the scaled feature vector is provided to the trained model to obtain a prediction.

5.3 Experiment Setup

In this section, I detail the experiment setup used to evaluate the validity and generality of Diaspore. Section 5.3.1 describes the cluster used for generating the dataset for building

and testing Diaspore. Section 5.3.2 lists the applications, collected metrics, and sources of interference used to build the dataset. Section 5.3.3 describes the data generation procedure along with the model building process. Lastly, Section 5.3.4 lays out the metrics used to guage the performance of Diaspore.

5.3.1 Experiment Testbed

I conduct a variety of experiments on the Lattice nodes of University of Calgary's ARC cluster [40]. Each node in Lattice has two sockets with Intel Xeon L5520 (Nehalem) 2.27 GHz quad-core processors. These 8 cores share 12 GB of RAM. The nodes in the Lattice cluster are connected by a 40 Gbit/s InfiniBand network. In my experiments, I configure a Spark executor with 5 processing cores and 10 GB of RAM per cluster node. I use the remaining processing cores and memory of the node for running other applications that generate interference on the node resources. All nodes in the cluster use Spark 2.2.0.

5.3.2 Experiment Factors

Table 5.1 lists the various experiment factors and their different levels. To study the effectiveness of my approach on a wide range of Spark applications, I consider 13 different applications spanning a variety of application types. These are the same applications that I use for evaluating Peridot.

I start by conducting small scale executions with 1 and 2 GB data on 1 and 2 executors, respectively. Each executor is allocated a separate cluster node in my experiments. These executions are mainly used for training the ML model because of their small execution time, which can reduce training effort and time. These small scale executions comprise nearly 88% of the application runs I carry out. The remaining executions are carried out on 50 and 100 GB data on 10 executors distributed over 10 cluster nodes. When studying the generalizability of the technique for large scale executions, I exclude these executions from the training set.

Table 5.1: Experiment factors and levels

Factor	**Levels**
Spark Applications	Wc, So, Lr, Nb, Co, Mm, Q9, Q15, Q26, Q52, Q64, Q70, Q78
Input Data Size (GB)	1, 2, 50, 100
# of Spark Executors	1, 2, 10
Type of Metrics	Application metrics, System metrics
Data Processing	Scaled metrics, Unscaled metrics
Granularity Level	Stage-level, Task-level
Prediction Threshold t	10%, 30%, 50%, 100%
Source of Interference	Processor, Memory
Interference Levels	Mild, Severe

I collect application and system metrics from all experiments and scale them with respect to the reference executions as described in Section 5.2.3. However, I also retain the unscaled metrics for comparing predictions with and without feature scaling.

To study Diaspore's ability to offer online predictions, I explore the accuracy of predictions for a stage when the prediction threshold t is varied. I vary t in 4 steps from 10% to 100%.

I focus on two types of interference resulting from contention for a cluster node's processor and memory. To create contention on these resources, I execute various Source of Interference (SoI) applications on the cluster nodes executing my Spark applications. To instantiate processor contention in the node, I use two applications, *Prime* and P_i. As described in Chapter 2, the first application computes prime numbers up to a specified value, whereas the second computes the value of π number up to a specified number of decimal points. Both applications have a very small memory footprint and are essentially meant to consume only the processor resources.

For memory interference, I use *RAMSpeed* [43] as a memory benchmark which reads and writes a specified amount of data from one memory location to another while performing different operations on the data such as scaling the data by an integer and adding data from two different memory locations. I also use another application, *MemArray*, for causing memory interference. This application merely creates and fills a 1 GB array in the node to consume a large chunk of memory. I use two different applications to induce the same type of resource interference so that I can have variety in the way a particular interference may be caused. To study the impact of a particular type of interference, I choose one SoI application that would cause the interference and execute it in parallel with the Spark application while monitoring Spark application metrics as well as system metrics.

I define interference intensity as the ratio of the mean task duration of an application stage with and without interference. For a given application, different SoI applications can cause different interference intensities. The interference intensity can be controlled by varying the number of parallel instances of SoI applications executing on a node. I vary the number of parallel SoI instances from 1 to 3. This results in a good spread of interference intensities for each application in my dataset. When the SoIs result in 1.5 times or more increase in a stage's mean task duration relative to the reference execution, the interference is categorized as severe. Otherwise, I consider the application to have encountered mild interference. In a few cases, 3 instances of SoIs do not result in severe interference. In such cases, I keep increasing the number of SoIs until the interference intensity is 1.5 or more. For example, I increased the number of SoIs for applications such as $Q26$ and Lr to up to 6 to observe severe interference. All data collected for Diaspore is available at [20].

5.3.3 Experiment Process

The primary experimental data comes from running each Spark application with 1 GB and 2 GB of data on 1 and 2 Lattice nodes, respectively. Each application is first executed on a single node without any interference with 1 GB of data to obtain the values of the

application metrics that can be achieved on an ideal system where there is no resource sharing among applications. As explained in Section 5.2, these reference executions are used for feature scaling of the metrics collected from other executions of the same application. For incorporating variations in the feature values during normal executions, I repeat each reference execution three times and collect average values of each feature. To observe the application behaviour under interference, one type of SoI is run at a time. This allows observation of the sole effect of one kind of resource contention during the experimentation. The number of instances of the interference running in parallel with the Spark application is controlled to include different intensity levels of interference in the dataset. If an application execution crashes due to very high interference, I exclude it from the final dataset because its failed tasks prevent me from gathering any metrics from them.

I also include executions for each Spark application on 2 GB of data executing on 10 cores distributed over 2 Lattice nodes, and 50 GB and 100 GB data on 50 cores distributed over 10 nodes. These executions help investigate if the technique is limited by the data size or the number of cores, and verifying if it can extrapolate to large scale applications without requiring prior data on such executions. From all executions, I gather application and system metrics only from the variable stages in each application to process the features and compile my dataset. This choice was made because variable stages constitute majority of the application time and hence can give best insights into the impact of interference on the application.

Once the dataset is finalized and the features that will be used for the model are selected, it is split into training, validation and test sets. The training set, as the name implies, is used by AutoML to train the model. AutoML uses the validation set to select the best model and the hyperparameters of the selected model. Once model parameters are selected, the model is evaluated on the test set. The training, validation and testing are part of the AutoML pipeline and hence are done automatically. By default, Google AutoML uses 80%

of the data for training, while validating it on 10% of the data and then eventually testing on the remaining 10% data.

5.3.4 Evaluation Metrics

To evaluate Diaspore's predictions, I report both the accuracy and the F-measure. As shown in Equation (5.1), accuracy quantifies the percentage of correct predictions out of all predictions made by the model. I also measure the model's precision and recall and use them to calculate the F-measure for the various scenarios I evaluate. As shown in Equation (5.2), precision represents the ratio of correctly identified samples (true positives) in one class to all data samples identified as that class. Equation (5.2) shows how I calculate precision for each of my classes, i.e., no interference, processor interference, and memory interference. Equation (5.3) shows recall for a class, which is the ratio of all the samples *correctly* identified for a class to all the samples that actually belong to that class. The F-measure, shown in Equation (5.4), is the harmonic mean of precision and recall and is a better performance measure for classifiers dealing with imbalanced datasets where one or more of the classes have only a limited number of samples. I aim for a model resulting in accuracy and F-measure values as close to 100% as possible. High accuracy denotes the ability of the model to make correct predictions and high F-measure verifies that the correctly predicted data samples are not influenced by any skew in data, i.e., all classes are equally benefiting from the high accuracy.

$$Accuracy = \frac{TruePositives}{TotalNumberofSamples} \times 100 \tag{5.1}$$

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives} \tag{5.2}$$

$$Recall = \frac{TruePositives}{TruePositives + FalseNegatives} \tag{5.3}$$

$$F - measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \times 100 \tag{5.4}$$

5.4 Results

Section 5.4.1, reports general observations from my experiments. Section 5.4.2 characterizes Diaspore when no application, data size, or executor settings are excluded from the training data. Section 5.4.3 studies the ability of Diaspore to offer predictions for large scale application executions that are not part of the training data. Section 5.4.4 characterizes the ability of the technique to provide predictions for an application missing in the training data. Section 5.4.5 shows the utility of using feature scaling. Diaspore's accuracy for various interference intensities is studied in Section 5.4.6. Section 5.4.7 studies the effectiveness of Diaspore when it is used to diagnose the root cause of performance interference in an online setting. Section 5.4.8 discusses salient findings of the study.

5.4.1 General Observations

Considering all scenarios, there are 351 experiments representing 95 hours of experimentation. The percentages of memory-interfered, processor-interfered, and non-interfered cases in my dataset are 39.8%, 42.7% and 17.5%, respectively. The number of processor-interfered and memory-interfered executions in my dataset is slightly different since I had to exclude some executions that failed because of extreme memory interference. The percentages of mild and severe interference in my dataset are 65.6% and 34.5%, respectively. On average, the task duration of an execution impacted by interference increased by 2.41 times the reference value. The largest inflation is for Wc where the task duration increases by a factor of 96.4 when subjected to $MemArray$. This indicates that anomalous application behavior, when left unchecked or unmitigated, can cause severe increases in execution time thereby motivating this work.

Figure 5.1 shows the execution time of applications when they are exposed to 3 instances of processor and memory interference along with the reference execution time of each application[1]. The values reported in this figure include memory interference induced by $RAMSpeed$ and processor interference caused by P_i. All applications are subjected to the same number and type of memory and processor SoI instances. However, from the figure, the impact of interference differs across applications. Memory interference has a more pronounced impact on execution time for a majority of applications. This suggests that the Lattice cluster is prone to encounter a memory bottleneck before a processor bottleneck.

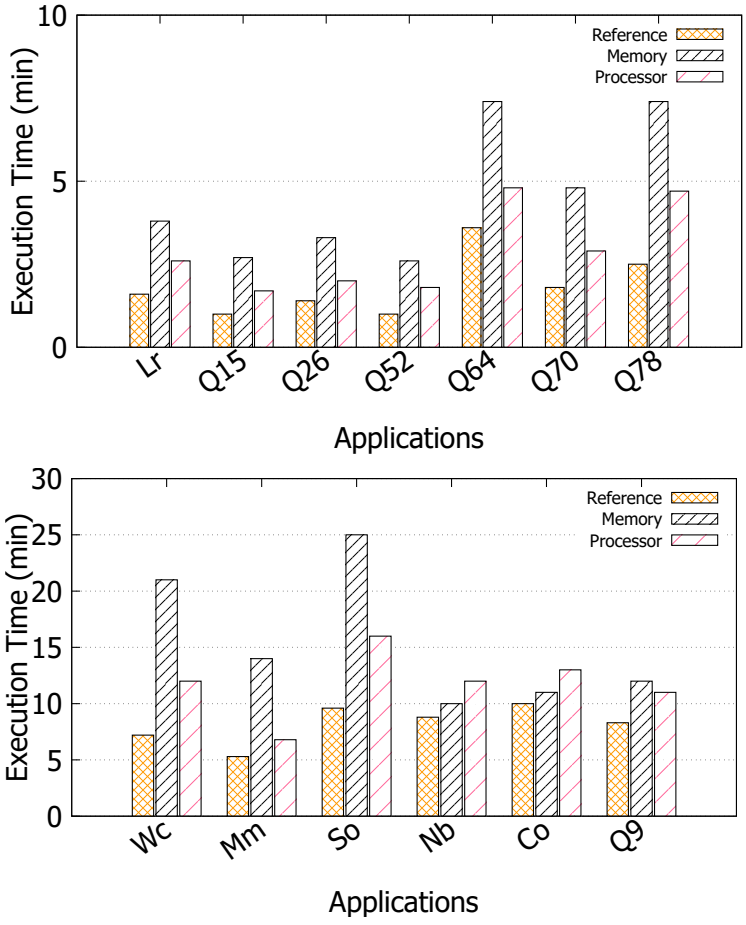

Figure 5.1: Execution times with different type of SoIs

Figure 5.2 shows the average increase in the task durations of the variable stages in my

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

applications relative to the reference as a function of the number of SoI instances. While the task durations increase with the number of SoI instances, the degree of impact varies across applications. Thus, simple techniques based on monitoring the number of SoI instances on cluster nodes and placing thresholds on task durations are unlikely to be effective in detecting interference and diagnosing its root cause. This motivates the need for ML techniques.

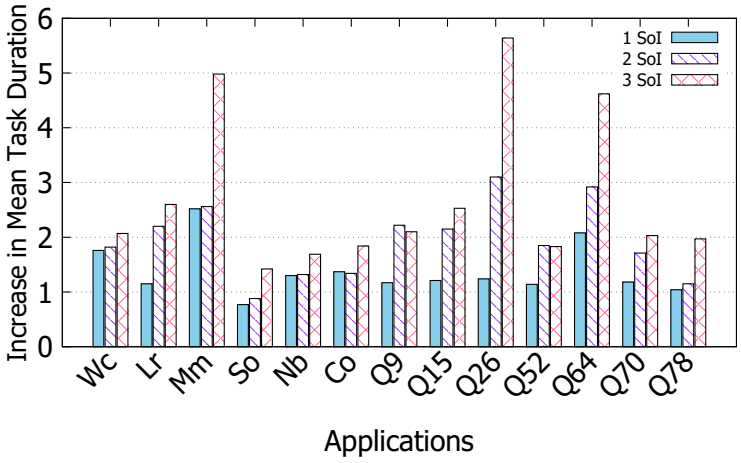

Figure 5.2: Increase in mean task duration with number of SoIs

When using the GCP's AutoML system, I assign 1 node-hour for training. If the model accuracy converges before that, the training process is stopped automatically. Typically, most models took the entire allocated time to train, with only 9% converging before 1 hour. In my study, AutoML always selected GBDT as the optimal classification model for this problem. AutoML tuned the hyperparameters of this model automatically.

5.4.2 Baseline Model

In the baseline experiments, the dataset presented to AutoML includes all applications, data sizes (1 GB to 100 GB), and processor counts (5 cores to 50 cores). The test set is a representative sample of the training set. This represents the ideal scenario for my global technique where the ML model has been trained on the applications and application configurations that it might have to make predictions for in the future.

I first consider a model selected and trained using task-level application and system metrics. This model yields an accuracy of 99.0% and an F-measure of 90.2%. However, when only the application metrics are considered, the model accuracy and F-measure drop to 60.0% and 80.9%, respectively. This suggests that task-level application metrics alone might not be very reliable in identifying the root cause of interference.

A possible reason for the poor performance with task-level metrics is the variability in task durations. For example, considering the first stage of the *So* application, task durations in the first stage vary by 56% even under no interference. Furthermore, task durations vary widely across different stages. Correlating these complex task duration patterns with different types of interference is likely to be easier when each application is modeled separately, as has been proposed in literature [13–15]. Such an approach increases training time since each application specific model needs to be trained under a wide range of data size and executor configurations. This motivates exploration of stage-level metrics.

Due to averaging, the stage-level metrics I consider have much less variability under any given level of interference intensity. Furthermore, previous work on Spark systems with no interference [44] has shown that the stage-level metrics show very little variability over different data size and executor configurations. This can be seen in Figure 5.3, which shows the mean task duration within a stage for selected stages in the reference and large scale runs of a representative subset of my applications. This lack of significant variability under no interference suggests that stage-level metrics might be more reliable for detecting abnormal behaviour should interference arise.

Based on these intuitions, I rebuild the model with stage-level application and system metrics. Using only application metrics, the model's performance improves significantly with an accuracy and F-measure of 97.3% and 97.1%, respectively. With both application and system metrics, I achieve 100% accuracy and F-measure. Thus, a single global model is very effective even when only application metrics are accessible, e.g., as in public-cloud based big

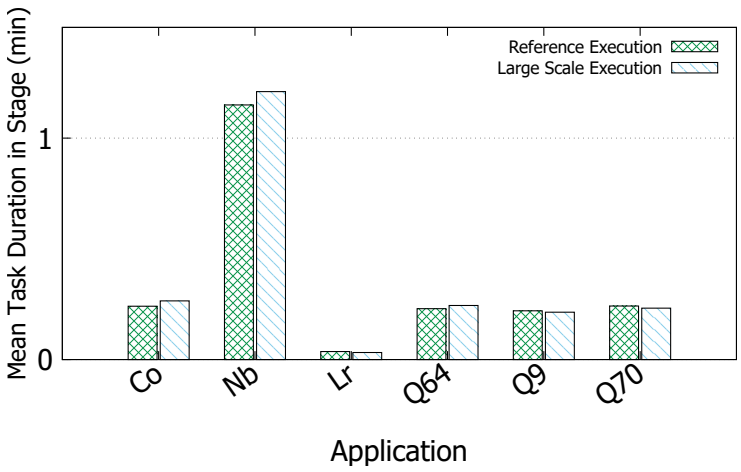

Figure 5.3: Stability of mean task duration of a stage

data systems.

While this baseline model performs well with stage-level metrics, I am interested in a model that can be generalized to large scale executions or to unseen applications without being trained on such data samples. I next investigate how Diaspore's stage-level model can generalize to new data sizes, executor settings and applications.

5.4.3 Performance on Large Scale Executions

This experiment is designed to evaluate Diaspore's ability to give accurate results on larger scale data sizes and executor settings. The model is trained with 1 GB and 2 GB datasets on 1 and 2 nodes, respectively. The trained model is then used to predict for large input data sizes, i.e., 50 GB and 100 GB. I note that the experiments with large input datasets use a different executor setting than the training data. Specifically, these experiments employ 10 executors (50 cores spread across 10 nodes).

Using only the application metrics in this scenario results in poor performance with an accuracy of 60.0% and an F-measure of 59.5%. However, combining both system and application metrics results in both the accuracy and F-measure being 83.3%.

Diaspore's ability to offer effective predictions based on only small scale executions re-

duces model training effort. Specifically, excluding the large scale experiments from the training set allows me to save nearly 45% on experimentation time and 90% on the processor cores needed for training. The training effort in terms of core-hours is reduced by approximately 80%. These results demonstrate the feasibility of using Diaspore in production environments to derive timely and accurate predictions.

5.4.4 Performance on Unseen Applications

Similar to the baseline experiments, I provide AutoML with both large scale and small scale datasets. However, I conduct a series of 13 experiments such that in each experiment one of my 13 applications is withheld from the training data. The test data only consists of records from that application. Thus, this experiment characterizes the ability of Diaspore's technique to generalize to unseen applications. Using application metrics performs poorly with 63.0% accuracy and F-measure. On the other hand, the complete feature set with application and system metrics yields very good performance with an accuracy of 83.4% and an F-measure of 88.8% on average. Figure 5.4 shows the accuracy for all 13 applications individually. From the figure, the accuracy ranges from 70.0% to 100%. $Q26$ is the hardest to predict when no relevant training samples are available. $Q70$, $Q78$ and $Q15$ give the best results with accuracies between $95\% - 100\%$. There is no particular trend observed with regards to certain types of applications being easier to predict than others. However, it is worth noting that the three query applications I achieve the highest accuracy on have some of the most complex inter-stage dependencies in my application set.

5.4.5 Effect of Feature Scaling

I next characterize the impact of the feature scaling described in Section 5.2. I use the unscaled dataset for comparison. With the scaled dataset, Diaspore uses an application's identity to either locate a historical reference execution or generate a reference execution to perform the scaling. Hence, for a fair comparison, I include the application name as an

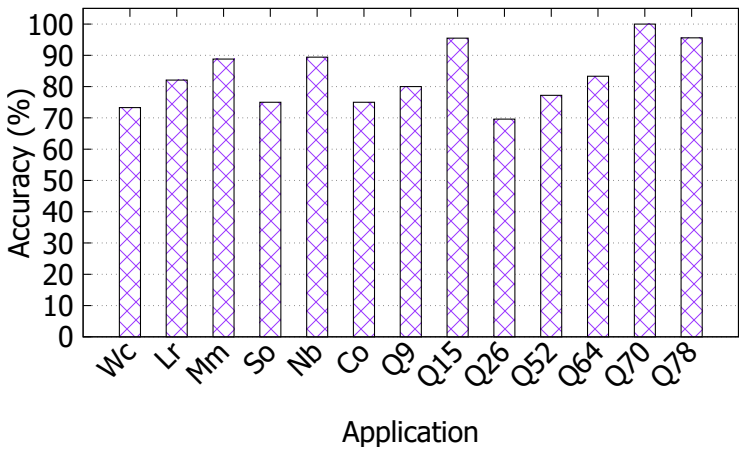

Figure 5.4: Accuracy for unseen applications

additional categorical feature when using the unscaled dataset. Models obtained by using an unscaled dataset without this feature resulted in very poor performance.

Figure 5.5 shows the results obtained with the application metrics feature set (A) and the application plus system metrics feature set (B). Both the training and tests datasets contain small scale and large scale executions. On average, the accuracy and F-measure of the baseline model trained with application and system metrics improves only marginally by 2.4% and 3.0%, respectively with scaled data. However, feature scaling is very effective in boosting performance when only application metrics are used. In this scenario, the scaled model is 18.0% more accurate than the unscaled model, with the F-measure being 8.2% better than the unscaled model. Thus, feature scaling is crucial for obtaining good performance in systems where only application metrics are available.

I also study the effect of feature scaling in aiding predictions for large scale executions and applications not recorded in the training data. While using both application and system metrics, feature scaling has no impact in improving the predictions for larger scale executions when such executions are excluded from training. However, the benefits are pronounced when using only application metrics. Scaling improves accuracy for this scenario by 13.0% and F-measure by 8.8%. For the unseen application scenario outlined in Section 5.4.4, scaling

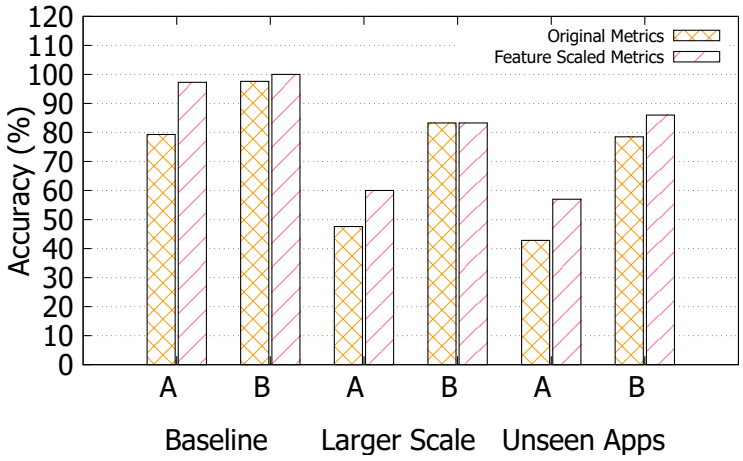

Figure 5.5: The effect of feature scaling, where A represents experiments on dataset with only application metrics and B represents experiments on dataset containing application and system metrics

increases accuracy and F-measure by 7.0% and 10.3%, respectively when both system and application metrics are used. The gain in accuracy and F-measure using scaled features is 14.0% and 16.9%, respectively when only application metrics are used. Table 5.2 summarizes the performance improvements obtained from feature scaling. These results suggest that scaling allows better generalization to unseen applications and is also crucial when only application metrics are available.

Table 5.2: Performance enhancement due to feature scaling (accuracy, f-measure)

Experiment	**Application Metrics**	**Application+System Metrics**
Baseline	18.0%, 8.2%	2.4%, 3.0%
Larger Scale	12.4%, 8.8%	0.0%, 0.0%
Unseen Applications	14.0%, 16.9%	7.0%, 10.3%

5.4.6 Impact of Interference Intensity

The previous experiments included both moderate and severe interference cases. I now investigate how Diaspore performs for each of these interference levels individually. For this experiment, I consider datasets consisting of all applications, data sizes, and executor settings. I compare results when moderate and severe records are eliminated from the dataset.

When considering only mild interference, the model based on only application metrics yields an accuracy of 81.9% and an F-measure 56.4%. Accuracy improves to 100% and F-measure increases to 98.0% when using both application and system metrics. Thus, mild interference is somewhat challenging to predict perfectly with just application metrics alone. When considering only severe interference, both approaches produce perfect accuracy, implying that extreme deviations in application behaviors are easier to detect and diagnose. Consequently, Diaspore allows cluster operators and users to effectively detect and mitigate severe performance issues.

5.4.7 Online Detection with Partial Stage Data

Finally, I study the effectiveness of online predictions using partial stage data collected from an executing application. For this experiment, I train the model on the small scale 1 GB and 2 GB experiments but test it on the larger scale 50 GB and 100 GB executions. I vary the prediction threshold t from 10% to 100%. Online predictions are particularly critical for large scale executions since they take a long time to complete and can waste system resources if interference is not detected early.

From Table 5.3, the accuracies and F-measures with partial stage data are comparable to those obtained with the full stage data. The lack of any significant difference in performance between the different t values can be explained as follows. In a large scale execution, even a small prediction threshold of t= 10% can correspond to a significant number of task completions. For example, Wc processing 100 GB input has 800 tasks in its variable stages,

which gives me 80 tasks to make a decision with t= 10%. This large number of tasks gives a sufficient window into the performance of the application for the technique to come up with an accurate prediction. In this case, the technique is able to predict interference after 12.5 minutes of the stage commencing instead of waiting for the full 66.0 minutes for the stage to complete. This result demonstrates Diaspore's effectiveness for online predictions.

Table 5.3: Performance evaluation with partial stage data

Threshold t (%)	**Accuracy (%)**	**F-measure (%)**
10	78.9	80.0
30	84.2	84.2
50	84.2	84.2
100	83.3	83.3

5.4.8 Discussion

My results suggest that a single global ML model trained with all applications and both small as well as large scale executions can be very effective if stage-level metrics are used as features. In an interference free system, the stage-level metrics I use as features show very little variability with respect to factors such as data size and executor settings. Thus, changes in these metrics can be used effectively to detect and diagnose execution time degradation due to interference.

Results show that scaling features with respect to their corresponding no-interference reference metrics allows highly accurate predictions to be made even when utilizing only application metrics. Thus, the scaling technique makes it feasible to deploy the model in environments which do not provide full visibility into system metrics, e.g., public-cloud based systems.

Both system and application metrics are needed for Diaspore to offer accurate predictions for large scale executions using a model trained only on smaller scale executions. This implies

drastically lower training effort and hence makes Diaspore attractive for deployments in real-world clusters where large scale training might not be feasible. The technique is also accurate when predicting for an application not present in the training dataset. In particular, feature scaling boosts the performance of the model for such scenarios. This result further illustrates Diaspore's practical applicability since real-world clusters encounter a significant fraction of new applications.

While effective for all interference intensities, Diaspore offers perfect predictions when trained to detect and diagnose severe interference scenarios. This allows cluster operators to quickly deploy mitigating strategies for severely impacted applications. Furthermore, Diaspore can offer accurate predictions based on a partial application execution thereby promoting agile diagnosis of interference. This can be particularly useful for improving resource utilization and avoiding user frustration while handling long running jobs facing interference.

5.5 Summary

My experimental results show that Diaspore is effective in detecting interference experienced by Spark applications regardless of their type or configuration. I have also demonstrated that Diaspore can be trained on very small scale executions, thus consuming very little time to generate the training data. Lastly, Diaspore can work with features extracted from stages that are not yet complete thereby providing quick predictions in an online setting.

Chapter 6

Conclusions and Future Work

This chapter summarizes the research done in this thesis and provides future research directions. Section 6.1 describes the outcomes of the two research problems I consider in this thesis. Limitations of these studies are discussed in Section 6.2. Finally, Section 6.3 discusses future research that can build on this work.

6.1 Summary

In this thesis, I tackle two important problems pertaining to modeling the performance of Apache Spark applications. I pursue the following two objectives:

1. Building a quick and lightweight modeling technique for predicting execution time of a Spark application.

2. Building a modeling technique that can quickly diagnose performance interference in a Spark platform.

To achieve these objectives, I devised analytical and ML models that can be built from a small subset of past executions. In both techniques, I focused on building models that can be generalized across several applications, data sizes, and core allocations. A summary of my contributions follows next.

My first contribution in this thesis is a lightweight, analytic execution time prediction approach for Spark applications called Peridot. 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. My thesis 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. The set of applications selected for my study 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 ML approaches to provide accurate predictions. I 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.

My second contribution is Diaspore, an interference detection and diagnosis technique for Spark applications. Although there is existing work related to this problem, there are a number of open issues that I explore in this research. Specifically, I systematically investigate

how to construct an ML-based model that will offer accurate predictions quickly, require very little training data, and generalize to scenarios not captured in the training data. Diaspore satisfies these requirements. I validate Diaspore on a real Spark cluster using a diverse set of applications and interference conditions.

Experimental results reveal several novel insights. First, I find that a stage-level featurization combined with feature scaling allows me to build a single compact model that can predict for many applications. Furthermore, this design choice allows the model to only rely on small scale executions for training. My results also indicate that the feature scaling allows Diaspore to accurately diagnose an unseen application that is not part of the training dataset. Experiments also reveal that sampling a small fraction of an application's execution is sufficient for detecting and diagnosing interference. Thus, Diaspore allows cluster operators to handle interference in an agile manner.

My results provide guidance on the use of application metrics alone for interference detection and diagnosis. This capability is critical in environments such as public clouds where operators of the big data service do not have access to system metrics that capture the impact of extraneous workloads on shared cluster node resources. Results suggest that one can obtain highly accurate predictions with just application metrics provided the model is trained with both small scale and large scale executions as well as all applications of interest.

6.2 Limitations

While both of my proposed techniques provide significant gains over existing approaches, they have limitations that can be addressed in future work. This section provides a discussion on these limitations.

1. Peridot shows good accuracy for predicting the execution time of applications on both the clusters I studied. However, the predictions are only valid for the

same cluster from which the reference executions are collected.

2. The predictions provided by Peridot are only valid when the structure of the input data, e.g., the number of columns in a matrix, remains the same as in the data used for the reference executions. This limits reusing the technique when the structure of the data changes.

3. To provide fast predictions while being lightweight, Peridot approximates an application's internal DAG structure. My results show that this approximation is effective even for applications with complex DAG structures. However, expanding the validation to more applications is needed to further understand the impact of the approximation.

4. For analytic simplicity, Peridot abstracts out several details related to Spark's partitioning behaviour. Specifically, it models variable and fixed stages separately and works on the assumption that the block size of each partition within a stage remains constant no matter the size of the input data. Thus, the time to process one partition, or a wave would remain the same for that stage for any input size. As my results suggest, this assumption is generally correct because it is the default Spark behavior. However, Peridot needs to be modified for scenarios where an application developer chooses to override the default Spark partitioning behaviour, e.g., statically fix the number of partitions in a stage.

5. Models proposed in this thesis are evaluated on different cluster platforms. However, each cluster has homogeneous cluster nodes. This is generally valid in real clusters where similar nodes are optimized for specific tasks, e.g., optimized for memory-intensive computations. However, the models are not devised to consider heterogeneous nodes in the cluster. Node heterogeneity can

affect the models, e.g., can affect the feature scaling if the reference execution is collected on a node different than the nodes that the incoming application is running on.

6. Diaspore works based on a data-driven ML technique. Such techniques can capture complex correlations between metrics and different types of interference. However, the accuracy of any data-driven technique depends on the fidelity of the data used for training it. Diaspore performs well because it was trained on a diverse set of applications and interference levels. In the absence of a good dataset, Diaspore's capacity to make accurate predictions is bound to decline.

7. Currently, Diaspore is designed to only classify memory and processor based interference. When an interference case is detected, the output root cause resource will only be limited to one of these two resources. If the model is exposed to instances of interference in other resources, say the network or the disk, Diaspore will not be able to detect them correctly. In fact, it may incorrectly try to classify them as a memory or processor interference.

6.3 Future Work

The Peridot technique described in this thesis can be expanded in multiple ways:

1. **Modeling complex stage behaviours** - As mentioned in the previous section, Peridot simplifies the behaviour of fixed stages with varying partition sizes. Although these stages generally do not contribute a lot to the overall application time, they do impact the prediction accuracy. Based on my initial analysis of these stages, the increase in mean wave time of these stages is not very linear with respect to input data size. Therefore, incorporating them

into the prediction process is not simple. An extension of Peridot can focus on studying such stages in depth and modeling the relationship between the increase in mean wave time with the increase of input data.

2. **Modeling with data characteristics** - Peridot works on the principle of mean wave times in a stage being similar despite changes in the number of cores or the input data size. However, an application stage's mean wave time can be impacted if the structure of the input data changes as shown in Section 4.4.1. Peridot works well as it samples the data for the reference executions from the actual input the application uses for its execution. Modeling the effect of data structure can reduce the number of reference executions required and can hence further speed up the prediction process.

3. **Generalizing predictions for other clusters** - As discussed previously, the reference executions are required to be collected on the same cluster platform for which the predictions will be provided. One interesting direction for future work can be using reference executions collected from one cluster environment to predict the execution time of applications on other cluster environments. Such a technique requires devising a transformation method that can map execution time behaviours observed on one cluster to the execution time behaviours on other clusters.

The Diaspore technique proposed in this thesis can be enhanced in the following directions:

1. **Detecting other types of interference** - Diaspore detects the presence of interference and classifies it as a memory-related or a processor-related interference. Other cluster resources, such as network or disk, can also suffer from interference and cause performance degradation for Spark applications.

Diaspore can be expanded by training it with interference scenarios for other cluster resources.

2. **Detecting multiple sources of interference** - Diaspore is currently built to identify one source of interference at a given time. However, in real systems, an application might face interference at more than one resource simultaneously. Therefore, an extension of this work can evolve Diaspore's ability to isolate multiple sources of resource interference simultaneously.

3. **Continual learning** - Another extension of this work would be to implement the ML model in a way for it to update itself. This means that the model should continually use the incoming data it sees during deployment to retrain itself. This is often used for keeping prediction models evolving and adaptive to the changes. While Diaspore works well overall, in Chapter 5, I show that when the test data is very different from the data used for training Diaspore, the prediction accuracy of Diaspore decreases slightly. Continual learning from incoming data can help Diaspore to evolve its model continuously to improve prediction.

4. **Interference mitigation** - An interference mitigation technique can be designed to use the output provided by Diaspore for reducing the negative effect of performance interference on Spark platforms. When an application is identified as suffering from interference and the main root cause resource is detected by Diaspore, mitigation policies can be activated to counter the interference. For example, applications suffering severely from interference can be terminated and quickly launched on another cluster node where interference for the resource is less likely.

Bibliography

[1] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, 2012.

[2] Apache spark monitoring and instrumentation. [Online]. Available: http://spark.apache.org/docs/latest/monitoring.html, 2017.

[3] Subhajit Sidhanta, Wojciech Golab, and Supratik Mukhopadhyay. Optex: A deadline-aware cost optimization model for spark. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 193–202. IEEE, 2016.

[4] Panagiotis Petridis, Anastasios Gounaris, and Jordi Torres. Spark parameter tuning via trial-and-error. In *INNS Conference on Big Data*, pages 226–237. Springer, 2016.

[5] Guolu Wang, Jungang Xu, and Ben 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)*, pages 586–593. IEEE, 2016.

[6] Giovanni Paolo Gibilisco, Min Li, Li Zhang, and Danilo Ardagna. Stage aware performance modeling of dag based in memory analytic platforms. In *Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pages 188–195. IEEE, 2016.

[7] Anastasios Gounaris, Georgia Kougka, Ruben Tous, Carlos Tripiana Montes, and Jordi

Torres. Dynamic configuration of partitioning in spark applications. *IEEE Transactions on Parallel and Distributed Systems*, 28(7):1891–1904, 2017.

[8] Shivaram Venkataraman, Zongheng Yang, Michael J Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378. USENIX Association, 2016.

[9] Muhammed Tawfiqul Islam, Shanika Karunasekera, and Rajkumar Buyya. dspark: Deadline-based resource allocation for big data applications in apache spark. In *13th IEEE International Conference on e-Science (e-Science)*, pages 89–98. IEEE, 2017.

[10] Rui Ren, Shuai Tian, and Lei Wang. Online anomaly detection framework for spark systems via stage-task behavior modeling. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*, pages 256–259, 2018.

[11] Kewen Wang, Mohammad Maifi Hasan Khan, Nhan Nguyen, and Swapna Gokhale. Modeling interference for apache spark jobs. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pages 423–431. IEEE, 2016.

[12] Siyang Lu, BingBing Rao, Xiang Wei, Byungchul Tak, Long Wang, and Liqiang Wang. Log-based abnormal task detection and root cause analysis for spark. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 389–396. IEEE, 2017.

[13] Siyang Lu, Xiang Wei, Bingbing Rao, Byungchul Tak, Long Wang, and Liqiang Wang. Ladra: Log-based abnormal task detection and root-cause analysis in big data processing with spark. *Future Generation Computer Systems*, 95:392–403, 2019.

[14] Meixiang Ding, Zhixiang Xiong, and Jian Yu. Anomaly detection based on job monitoring metrics in distributed system. In *2018 International Conference on Security, Pattern Analysis, and Cybernetics (SPAC)*, pages 182–186. IEEE, 2018.

[15] Ahmad Alnafessah and Giuliano Casale. Artificial neural networks based techniques for anomaly detection in apache spark. *Cluster Computing*, 23(2):1–16, 2019.

[16] Laiping Zhao, Yiming Li, and Francoise Fogelman-Soulié. Effective straggler mitigation with cross-layer interference-aware optimization. In *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 177–182. IEEE, 2019.

[17] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards automated slos for enterprise clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 117–134, Savannah, GA, 2016. USENIX Association.

[18] Sarah Shah, Yasaman Amannejad, Diwakar Krishnamurthy, and Mea Wang. Peridot data set. [Online]. Available: https://github.com/Yasaman-A/Peridot, 2020.

[19] Xin He, Kaiyong Zhao, and Xiaowen Chu. Automl: A survey of the state-of-the-art. *arXiv preprint arXiv:1908.00709*, 2019.

[20] Sarah Shah, Yasaman Amannejad, and Diwakar Krishnamurthy. Diaspore data set. [Online]. Available: https://github.com/Yasaman-A/Diaspore, 2020.

[21] Raghunath Nambiar and Meikel Poess. TPC-DS Benchmark. [Online]. Available: http://www.tpc.org/tpcds/.

[22] Tuning Spark. [Online]. Available: https://spark.apache.org/docs/latest/tuning.html#garbage-collection-tuning.

[23] Minseo Kang and Jae-Gil Lee. Effect of garbage collection in iterative algorithms on spark: an experimental analysis. *The Journal of Supercomputing*, pages 1–15, 2020.

[24] Wanling Gao, Yuqing Zhu, Zhen Jia, Chunjie Luo, Lei Wang, Zhiguo Li, Jianfeng Zhan, Yong Qi, Yongqiang He, Shiming Gong, Xiaona Li, Shujie Zhang, and Bizhu Qiu. BigDataBench: a Big Data Benchmark Suite from Web Search Engines. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 488–499. IEEE, 2013.

[25] Apache spark mllib. [Online]. Available: https://spark.apache.org/mllib/, 2017.

[26] Abhishek Arora. Scalable-Matrix-Multiplication-on-Apache-Spark. [Online]. Available: https://github.com/Abhishek-Arora/Scalable-Matrix-Multiplication-on-Apache-Spark.

[27] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019.

[28] AutoML. [Online]. Available: https://cloud.google.com/automl/docs, 2019.

[29] Kewen Wang and Mohammad Maifi Hasan 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 Conference on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on High Performance Computing and Communications*, pages 166–173, 2015.

[30] Adrian Daniel Popescu. *Runtime Prediction for Scale-Out Data Analytics*. PhD thesis, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, 2015.

[31] Zemin Chao, Shengfei Shi, Hong Gao, Jizhou Luo, and Hongzhi Wang. A gray-box performance model for apache spark. *Future Generation Computer Systems*, pages 58–67, 2018.

[32] Yasaman Amannejad, Sarah Shah, Diwakar Krishnamurthy, and Mea Wang. Fast and lightweight execution time predictions for spark applications. In *IEEE 9th International Conference on Cloud Computing (CLOUD 2019)*, pages 1–3. IEEE, 2019.

[33] Rekha Singhal and Praveen 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*, pages 131–146, 2017.

[34] Friedrich Pukelsheim. *Optimal design of experiments*, volume 50. SIAM, 1993.

[35] Apache spark job scheduling. [Online]. Available: https://spark.apache.org/docs/latest/job-scheduling.html, 2017.

[36] Thanasis Theocharis Stathis Maroulis, Nikos Zacheilas and Vana Kalogeraki. Fast. efficient performance predictions for big data applications. In *2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)*, pages 126–133, May 2019.

[37] Jinbae Lee, Bobae Kim, and Jong-Moon Chung. Time estimation and resource minimization scheme for apache spark and hadoop big data systems with failures. *IEEE Access*, 7:9658–9666, 2019.

[38] Luciano Baresi, Alberto Leva, and Giovanni Quattrocchi. Fine-grained dynamic resource allocation for big-data applications. *IEEE Transactions on Software Engineering*, 2019.

[39] Ahmad Alnafessah and Giuliano Casale. Track-plus: Optimizing artificial neural networks for hybrid anomaly detection in data streaming systems. *IEEE Access*, 8:146613–146626, 2020.

[40] Advanced research computing cluster. [Online]. Available: https://hpc.ucalgary.ca, 2018.

[41] Danilo Ardagna, Enrico Barbierato, Athanasia Evangelinou, Eugenio Gianniti, Marco Gribaudo, Túlio B. M. Pinto, Anna Guimarães, Ana Paula Couto da Silva, and Jussara M. Almeida. Performance prediction of cloud-based big data applications. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, pages 192–199, New York, NY, USA, 2018. ACM.

[42] Meikel Poess, Raghunath Othayoth Nambiar, and David Walrath. Why you should run tpc-ds: A workload analysis. In *VLDB*, volume 7, pages 1138–1149, 2007.

[43] Rhett M. Hollander and Paul V. Bolotoff. Ramspeed, a cache and memory benchmarking tool. [Online]. Available: http://alasir.com/software/ramspeed, 2002.

[44] Sarah Shah, Yasaman Amannejad, Diwakar Krishnamurthy, and Mea Wang. Quick execution time predictions for spark applications. In *Proceedings of the International Conference on Network and Service Management (CNSM 2019)*. IEEE, 2019.

Appendix A

Permission of Copyright

IEEE copyright - Author's published papers in [32, 44] are IEEE copyrighted materials. The copyright permissions for the papers used in this thesis is obtained from IEEE's RightsLink and are attached in next three pages.

Notice of the IEEE Copyright

> In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of University of Calgary's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html.