Autonomic Performance Optimization of Big Data Workloads

by

Mikhail Genkin, B.Sc., M.Sc.

A thesis submitted to the
Faculty of Graduate and Postdoctoral Affairs
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

Ottawa-Carleton Institute for Electrical and Computer Engineering
Department of School of Computer Science
Carleton University
Ottawa, Ontario
January, 2021

©Copyright
Mikhail Genkin, 2021

The undersigned hereby recommends to the
Faculty of Graduate and Postdoctoral Affairs
acceptance of the thesis

**Autonomic Performance Optimization of Big Data
Workloads**

submitted by **Mikhail Genkin, B.Sc., M.Sc.**

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

Professor Tamer Özsu, External Examiner (**PHD only**)

Professor Frank K. Dehne, Thesis Supervisor

Professor Anil Maheshwari, Graduate Director,
School Of Computer Science

Ottawa-Carleton Institute for Electrical and Computer Engineering
Department of School of Computer Science
Carleton University
January, 2021

Abstract

The big data software stack based on Apache Spark and Hadoop has become mission critical in many enterprises. Performance of Spark and Hadoop jobs depends on a large number of configuration settings. Manual tuning is expensive and brittle. There have been prior efforts to develop on-line and off-line automatic tuning approaches to make the big data stack less dependent on manual tuning. These, however, demonstrated only modest performance improvements with very simple, single-user workloads on small data sets. In the more traditional "small data" space, research into autonomic systems development produced workload prediction and re-configuration techniques based on past workload history analysis. These have only limited applicability in the big data space, where workloads tend to be less repetitive. This thesis presents KERMIT - the autonomic architecture for big data capable of automatically tuning Apache Spark and Hadoop on-line, and achieving performance results 30% faster than rule-of-thumb tuning by a human administrator and 92% as fast as the fastest possible tuning established by performing an exhaustive search of the tuning parameter space. KERMIT can detect important workload changes with 99% accuracy, and predict future workload types with 96% accuracy. It is capable of identifying and classifying complex multi-user workloads without being explicitly trained on examples of these workloads. It does not rely on the past workload history to predict the future workload classes and their associated performance. KERMIT can identify and learn new workload classes, and adapt to workload drift, without human intervention. This thesis presents three new machine learning algorithms - a low-overhead on-line search algorithm, a statistical ensemble algorithm for real-time change detection, and a new, advanced, zero-shot machine learning algorithm for identification of unseen hybrid classes.

I dedicate this thesis to my wonderful family - my mother Ida, my wife Cynthia, and my children Anna, Peter, and Alexander. Their encouragement, support, patience and understanding kept me going and focused through so many late nights and weekends spent in front of the computer. I also dedicate this thesis to the memory of my father Boris, whose accomplishments were always a great inspiration to me.

Acknowledgments

I would like to acknowledge the direction and contributions provided by my thesis supervisor Prof. Frank Dehne. His wisdom and encouragement were critical in helping keep this thesis on track. I would also like to thank, and acknowledge the contributions made by the master and bachelor students who collaborated with me on the KERMIT project - Maria Pospelova, Pablo Nararro, Yabing Chen, Siyu Zhou, Dave Nelson, Alexander Trostanovsky, and Anousheh Shahmirza. Their hard work really enhanced the quality of this study and was instrumental in its timely completion.

Preface

This thesis is in "integrated article format". Several chapters are based on published papers, conference proceedings, or papers awaiting publication:

- Chapter 1 is a combined introduction that weaves together subsequent research papers organized into chapters, and outlines the over-arching vision. This chapter is entirely my work.
- Chapter 2 gives an overview of the big data technology stack and presents justification of the technology choices used for the research studies presented in the subsequent chapters. Material in this chapter is meant to compliment technology-focused discussions in the subsequent chapters. This chapter is entirely my work.
- Chapter 3 delves into the scheduling aspect of the big data stack, and describes the most import scheduling algorithms in that space. This chapter also discusses pre-emption. Resource scheduling and pre-emption are discussed in the subsequent research investigations that form this thesis. The purpose of this chapter is to provide the technological context for them. This chapter is entirely my work.
- Chapter 4 introduces the machine learning techniques used in the subsequent research investigations presented in this thesis. This chapter is entirely my work.
- Chapter 5 presents the first investigation focusing on on-line automatic tuning of the YARN container density for two different analytic frameworks. It is based on papers published in the 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems [13] and represents joint work with Prof. Frank Dehne, Maria

Pospelova and Iris Chen. My contributions to this chapter/paper include developing the overall concept, designing the KERMIT-to-resource manager integration, setting up the big data cluster and the development environment, supervising Maria's implementation work, developing benchmarking scripts for data collection, leading interpretation of the collected data, and leading the manuscript preparation for the paper. Design of the Explorer algorithm was a team effort with contributions from me, Prof. Dehne and Maria. Prototype code was written by Maria and Iris, under my supervision. Experimental data were collected by Iris.

- Chapter 6 presents machine-learning-based workload classification using container performance patterns. This work was presented at the IEEE Big Data 2018 Conference, 2018 International Symposium on Benchmarking, Measuring and Optimizing (Bench' 18), and was published in Lecture Notes in Computer Science volume 11459 [12]. It represents joint work with Prof. Frank Dehne, Pablo Navarro and Siyu Zhou. My contributions to this paper/chapter include developing the overall concept, supervising the design and development of KERMIT extensions by Pablo and Siyu, designing and developing the machine learning classifiers, leading data analysis and manuscript preparations. Experimental data were collected by Siyu.
- Chapter 7 focuses on classifying not only workloads, but changes in workload characteristics (workload transitions), developing a 'workload language', and using an LSTM to predict future workload types. This work was presented at the 2019 IEEE International Conference on Big Data in Los Angeles and was published in the IEEE conference proceedings [11]. It represents joint work with Prof. Frank Dehne. My contributions to this paper included developing the overall concept for the paper, designing and developing prototypes for the machine learning components, collecting experimental data and leading the manuscript preparation. Prof. Dehne contributed to the overall paper concept, analysis and manuscript preparation.
- Chapter 8 presents a new technique for classifying unseen multi-user workloads. It presents a new advanced zero-shot learning algorithm capable of identifying unseen multi-user workloads after being trained on seen single-user workloads. This work was presented at the 2020 IEEE International Conference on Big Data

and is in the process of being published in the IEEE conference proceedings [14]. This chapter/paper is entirely my work.

- Chapter 9 builds on the research investigations presented in the previous chapters and describes the autonomic architecture for big data performance optimization. This work is currently being prepared for submission to a journal that specializes in similar topics. This work is joint work with Anousheh Shahmirza. My contributions to this work included developing the overall concept, researching previous work, designing and documenting the overall KERMIT architecture, designing the KERMIT plug-in algorithm and the off-line machine learning pipeline, developing the code for the key integration points, supervising clustering analysis work performed by Anousheh, and leading the manuscript preparation. Anousheh contributed comparative analysis of clustering algorithms used to identify different workload types - under my supervision.
- Chapter 10 summarizes key findings of the thesis, discusses the implications for the architecture of the big data stack, and proposes future work. This chapter is entirely my work.

Bibliography:

[13] M. Genkin, F. Dehne, M. Pospelova, Y. Chen, and P. Navarro, "Automatic, on-line tuning of yarn container memory and cpu parameters," in 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems, pp. 317–324, IEEE, 2016.

[12] M. Genkin, F. Dehne, P. Navarro, and S. Zhou, "Machine-learning based Spark and Hadoop workload classification using container performance patterns," in C. Zheng and J. Zhan (Eds) 2018 International Symposium on Benchmarking, Measuring and Optimizing (Bench' 18), vol. 11459, pp. 118 - 130, Springer, 2019.

[11] M. Genkin and F. Dehne, "Autonomic Workload Change Classification and Prediction for Big Data Workloads," in *2019 IEEE Conference on Big Data (Big Data)*, Los Angeles, CA, USA., pp. 2835 - 284, IEEE, 2019.

[14] M. Genkin, "Zero-shot machine learning technique for classification of multi-user big data workloads," in *2020 IEEE International Conference on Big Data (Big Data)*, Atlanta, GA, USA., pp. in print, IEEE, 2020.

Table of Contents

Abstract	iii
Acknowledgments	v
Preface	vi
Table of Contents	ix
List of Tables	xiv
List of Figures	xv
1 Introduction	1
1.1 The Problem	1
1.2 How To Read This Thesis	2
1.3 Key Research Questions	3
1.4 Completed Research Investigations	4
1.4.1 Automatic, On-line Tuning of YARN Container Density . . .	4
1.4.2 Machine Learning Based Spark and Hadoop Workload Classifi- cation Using Container Performance Patterns	6
1.4.3 Autonomic Workload Change Classification and Prediction for Big Data Workloads	6
1.4.4 Zero-shot Machine Learning Technique for Classification of Hy- brid Classes	7
1.4.5 Autonomic Architecture for Big Data	8
1.4.6 Conclusion	8

2	Big Data Architecture	9
2.1	Architecture Overview	9
2.2	Analytic Frameworks	12
2.2.1	Hadoop MapReduce	12
2.2.2	Apache Spark	14
2.3	Resource Managers and Containers	15
2.4	Conclusion	18
3	Big Data Resource Scheduling	20
3.1	Resource Managers and Schedulers	20
3.1.1	The FIFO Scheduler	21
3.1.2	The Capacity Scheduler	22
3.1.3	The Fair Scheduler	23
3.2	Pre-emption	25
3.3	Conclusion	25
4	Machine Learning and Autonomic Computing	27
4.1	Machine Learning Types and Use Cases	28
4.1.1	Popular SML Algorithms	28
4.1.2	Popular UML Algorithms	33
4.2	Statistical Classifiers	35
4.3	DecisionTree and RandomForest Algorithms	36
4.4	Artificial Neural Networks	38
4.4.1	ANN Architecture	38
4.4.2	Training ANN	42
4.5	Recurrent Neural Networks and Long Short-Term Memory Networks	44
4.6	Applying Machine Learning Techniques - Methodology	46
4.7	Model Selection and Evaluation	47
4.8	Advanced Machine Learning	49
4.9	Combining Machine Learning Techniques	52
4.10	Conclusion	52
5	YARN Automatic Tuning	54
5.1	Problem	54
5.2	Contribution	55

5.3	Previous Work	55
5.4	KERMIT Automatic Tuning Architecture	58
5.4.1	YARN Architecture Review	59
5.4.2	KERMIT Architecture	59
5.5	Evaluation Methodology	64
5.6	Results	66
5.7	Conclusion	74
6	Workload Classification	76
6.1	Problem	76
6.2	Contribution	77
6.3	Previous Work	78
6.4	Evaluation Methodology	81
6.4.1	Container Performance Metrics	81
6.4.2	Parameter Settings	82
6.4.3	Hardware and Software	84
6.5	Results	85
6.5.1	Steady State Workload Characteristics	85
6.5.2	Dynamic Workload Characteristics - Workload Transitions	85
6.5.3	Identifying and Classifying Workloads	87
6.5.4	Detecting Workload Transitions	89
6.5.5	Relative Value and Importance of Container Performance Metrics	93
6.6	Implications for the Development of On-Line Automatic Tuning Systems	94
6.7	Conclusion	95
7	Autonomic Workload Prediction	97
7.1	Problem	97
7.2	Limitations of Previous Approaches	98
7.3	Contribution	98
7.4	Previous Work	99
7.5	Workloads and Workload Changes in Big Data Applications	103
7.6	Architecture	105
7.7	Evaluation Methodology	110
7.7.1	Parameter Settings	112
7.7.2	Hardware and Software	112

7.8	Results	113
7.8.1	Change Detection	113
7.8.2	Classifying Workloads	114
7.8.3	Classifying Workload Transitions	116
7.8.4	Predicting Performance	117
7.9	Conclusion	119
8	Classifying Multi-User Workloads	120
8.1	Introduction	120
8.1.1	Background	120
8.1.2	Problem	121
8.1.3	Limitations of Previous Approaches	123
8.1.4	Contribution	124
8.2	Previous Work	124
8.3	The Big Data Workload Spectrum	125
8.4	Classifying Hybrid Classes	130
8.4.1	The Semantic Space	131
8.4.2	The Class Prototyping Function	132
8.4.3	The Synthesizing Function	133
8.5	The ZSL Classifier Architecture	134
8.6	Evaluation Methodology	137
8.7	Results	139
8.8	Conclusion	143
9	Autonomic Architecture for Big Data	145
9.1	Problem	145
9.2	Limitations of Previous Approaches	146
9.3	Contribution	146
9.4	Previous Work	147
9.5	Autonomic Architecture for Big Data Workload Optimization	151
9.5.1	Key Concepts and Terminology	151
9.5.2	Autonomic Architecture Principles	152
9.5.3	The KERMIT Architecture	153
9.6	The KERMIT On-line Sub-System Architecture	155
9.6.1	Workload Monitoring	155

9.6.2	Workload Classification, Prediction, and Optimization	156
9.7	The KERMIT Off-Line Sub-System Architecture	160
9.7.1	Workload Discovery, Characterization, and Drift Detection	161
9.7.2	Automated Classifier Training	165
9.8	Conclusion	167
10	Conclusion	169
10.1	Summary	169
10.2	Implications for the Big Data Stack Architecture	171
10.3	Future Work	173
	List of References	176

List of Tables

5.1	KERMIT automatic tuning results, compared to the baselines, for the TPCx-HS benchmark at various data scales.	71
6.1	Workloads, workload transitions, and benchmarks.	83
7.1	Workload and workload transition classes and labels.	106
7.2	Observation window features and their descriptions.	106
8.1	Workloads and benchmarks used to collect observation window data for the WorkloadSynthesizer training and validation sets.	137
8.2	Multi-user (hybrid) and single-user (pure) workload scenarios and benchmarks used to collect observation window data for the WorkloadSynthesizer testing set.	139

List of Figures

2.1	Apache Hadoop ecosystem and logical architecture in 2014.	10
2.2	Apache Hadoop ecosystem and logical architecture in 2019.	11
2.3	Apache Spark cluster architecture [49]. Figure shared under open-source Apache License 2.0.	14
2.4	Tez integration with YARN.	15
2.5	Apache Hadoop YARN architecture [17]. Figure shared under open- source Apache License 2.0.	16
2.6	Apache Spark Integration With Kubernetes [30]. Figure shared under open-source Apache License 2.0.	17
2.7	Docker component flow [9]. Figure shared under open-source Apache License 2.0.	18
3.1	The YARN architecture with focus on the scheduler and the application master.	21
3.2	The YARN FIFO scheduler algorithm.	22
3.3	The YARN capacity scheduler algorithm.	23
3.4	The YARN fair scheduler algorithm.	24
4.1	K-Means vs. DBSCAN clustering examples. Image from [46] is shared under open-source MIT license.	35
4.2	The decision tree logical structure.	36
4.3	Architecture of a simple artificial neural network.	39
4.4	Architecture of the artificial neuron.	40
4.5	RNN architecture.	45
4.6	LSTM architecture. This figure is Figure 1 from [16].	46
4.7	Bias-variance trade-off and model complexity illustrated.	48
4.8	A ZSL example - animals with attributes. This graphic is Figure 1 from the research paper by Lampert et al. [31].	51

5.1	Apache Hadoop YARN architecture. Figure shared under open-source Apache License 2.0 [17].	60
5.2	KERMIT architecture.	61
5.3	How KERMIT integrates with YARN.	62
5.4	Normalised performance summary for the Apache Hadoop and Spark benchmarks.	67
5.5	Data for the best possible tuning for TeraSort at the 500 GB data scale.	68
5.6	Data for the best possible tuning for TPCx-HS at the 500 GB data scale.	68
5.7	Data for the best possible tuning for SMB.	69
5.8	The Explorer algorithm optimizing the container durations for the SMB workload.	70
5.9	The Explorer algorithm optimizing the container memory allocations for the SMB workload.	70
5.10	Comparison of KERMIT on-line automatic tuning with the best possible tuning for TPCx-HS at different data scales.	72
5.11	Performance of KERMIT on-line automatic tuning for TPCx-HS appears to show linear increase with increasing data volume.	73
5.12	The KERMIT efficiency (percentage of the best possible tuning performance achieved by KERMIT) at different data scales.	73
6.1	Test-bed topology.	84
6.2	Radar chart showing average Hadoop and Spark workload metric values.	86
6.3	Radar chart showing average Hadoop and Spark workload transition values.	87
6.4	Workload classification accuracy for common machine-learning algorithms.	88
6.5	Impact of using different container performance statistics on the classification accuracy of the Random Forrest algorithm.	89
6.6	Container performance metrics observed for Hadoop WordCount-TeraSort-K-Means job flow, using a 2 min observation window and 2 TB data size.	90
6.7	Container performance metrics observed for multi-user Spark workload with batch and interactive query components, using a 30 sec observation window, 2 GB data.	91
6.8	Workload transition detection accuracy for different container performance metrics.	93

6.9	Possible automatic tuning system architecture.	95
7.1	Theoretical relationship between workloads, workload transitions, observation windows, and analytic windows.	108
7.2	High-level architecture for on-line automatic tuning system for big data.	109
7.3	Test-bed topology.	113
7.4	ChangeDetector ROC for different thresholds, and equal weights.	115
7.5	ChangeDetector summary of key classifier performance metrics.	115
7.6	Average Accuracy, PPV and FP Rate achieved for the WorkloadClassifier.	116
7.7	Average Accuracy, PPV and FP Rate achieved for the TransitionClassifier.	117
7.8	WorkloadPredictor summary of accuracy by prediction horizon.	118
8.1	How single-user feature vectors combine to form a multi-user feature vector. HDRED and SPSTAGE are defined in [11]. The dashed arrows indicate the aggregation of the observation windows.	126
8.2	Multi-dimensional workload spectrum formed by workload classes defined in [11].	129
8.3	How hybrid classes are formed from pure classes.	131
8.4	Architecture of the ZSL workload classifier.	135
8.5	Cluster architecture.	138
8.6	The effect of increasing the number of generated synthetic hybrid instances on the accuracy of hybrid workload classification of the ZSL workload classifier.	139
8.7	Comparison of feature vector values for the HDMAP workload phase generated by the Hadoop TeraSort and K-Means jobs, and the TeraSort HDRED workload.	141
8.8	Accuracy, PPV, and FP Rate for the WorkloadClassifier on pure and hybrid workload classes.	141
8.9	Scalability of the WorkloadSynthesizer component.	142
9.1	The MAPE-K reference architecture for autonomic systems proposed by IBM. This figure was constructed based on the reference architecture description in [24].	148
9.2	Mapping the KERMIT sub-systems and components onto the MAPE-K reference architecture.	153

9.3	The high-level KERMIT components shown in relation to the resource manager and the big data cluster.	154
9.4	Logical architecture of the KERMIT knowledge base.	157
9.5	The high-level steps in the off-line sub-system processing pipeline. . .	161
9.6	Workload discovery performance for clustering algorithms.	163
9.7	Entity-relationship model of the WorkloadDB schema.	164
10.1	Possible integration among KERMIT, the resource manager and the analytic frameworks.	172
10.2	KERMIT positioning within the logical big data stack.	173

Chapter 1

Introduction

1.1 The Problem

Big data analytics has emerged as one of the most important computing trends in recent years. The main factor that separates the big data space from the more traditional "small data" space is, of course, the size of the data. Big data applications need to perform complex queries on Terabytes (TB) to Petabytes (PB) of data. A big data analytic job can take hours, to days, to weeks to run, depending on the size of the data actually being scanned, and the complexity of the query being performed. If the job takes too long to run, the value of the information it returns may degrade. For example, if we are running a job intended to analyse the impact of a political debate on voter sentiment by analysing Twitter feeds, and the job takes several days to run, the insights it produces may be stale because voter attention and focus may change during this time. Clearly, performance is a key aspect of any big data application or system.

Performance of big data applications is strongly influenced by many configuration settings, and is heavily dependent on tuning. Manual tuning of big data applications, such as Hadoop MapReduce or Apache Spark jobs, involves changing many tuning parameters. The big data technology stack defines several layers - the storage layer, the analytic framework layer, and the resource management layer. Performance of each architectural layer is governed by a specific set of configuration files. For example, tuning the performance of an Apache Hadoop MapReduce job can involve adjusting more than two dozen parameters located in several configuration files. Understanding the impact of all of these parameters requires developing in-depth knowledge of many disparate technologies. The administrator would need to understand the workings of

the MapReduce framework, the HDFS file system, and the YARN resource manager. Each of these technologies presents a steep learning curve. Learning how to tune the entire big data stack is extremely challenging.

Optimal tuning configuration often has to be determined experimentally. The administrator must adjust the configuration values of the tuning parameters one at a time, and execute the job on a representative set of data in the pre-production environment. Considering that the typical scale of the data in the big data space ranges from several TB to PB, each experiment could take many hours to run. Manual tuning also often results in sub-optimal and brittle job performance, because the parameter values that are optimal for one job may not be well suited to another.

As the size of the data grows, the application may need to be re-tuned. The size of the data available in the pre-production environment will rarely be the same as the size of the data in the production environment. The number of processing nodes in the pre-production environment will rarely be the same as the number of processing nodes in the production environment. Thus, a job that was tuned manually in the pre-production environment is not guaranteed to run fast in the production environment.

Manual tuning of big data applications is very expensive. Teaching system administrators how to tune big data applications is also very expensive. If we could find a way to make the big data stack more autonomic - capable of tuning itself - this would resolve a major pain point for the industry, and remove an important barrier to the wider adoption of big data analytics.

1.2 How To Read This Thesis

This thesis is organized as a sequence of research investigations intended to find answers to several key research questions. These research investigations are supported by several background chapters intended to describe the relevant technology landscape. The focus of each chapter is briefly described in the paragraphs below.

Chapter 2 presents an architectural overview of the big data technology stack. The intent behind this chapter is to provide the overall technology context for the research investigations presented in the later chapters.

Chapter 3 expands on the job scheduling aspects of the big data stack, describing the scheduler architecture, the most important scheduling algorithms, and the concept

of pre-emption. These concepts are discussed in the research investigations presented in subsequent chapters.

Chapters 5, 6, 7, and 8 present completed and published research investigations. These chapters are based on published research papers, and are meant to be self-contained. Each of these chapters includes the detailed statement of the problem, the statement of contribution, the summary of previous research, and sections describing the approach, the results, and the conclusions for that investigation.

Chapter 9 is an architecture-focused chapter that brings together the findings presented in the previous chapters into a coherent autonomic solution. This chapter is also organised as a research paper, and is also self-contained. It is currently being prepared for submission to a suitable scientific journal.

Finally, Chapter 10 presents the overall conclusion for this thesis and discusses the potential implications for the evolution of the big data stack and related technologies. This chapter also includes a summary of the proposed future work.

Sections below begin with a more detailed discussion of the key research questions that form this thesis. Following this discussion each completed or proposed research investigation/chapter is introduced in more detail.

1.3 Key Research Questions

The over-arching research question behind this thesis can be stated as: *"Can we make the big data technology stack autonomic - capable of tuning itself, rather than relaying on a human system administrator to do it?"* This, however, is a very broad question. In order to answer this question it needs to be broken down into a sequence of narrower questions that can be answered by more manageable research investigations:

1. *Can we automatically tune the key configuration parameters for the big data frameworks on-line?* Before delving into the more complex situations addressed by the subsequent questions it is important to demonstrate that it is feasible, in principle, to automatically tune key big data technologies on-line using known and relatively simple workloads.
2. *Can we accurately classify workloads, and detect changes in important workload characteristics?* It is impossible to tune a system efficiently in a workload-agnostic manner. An autonomic system must have the ability to accurately identify and classify workloads so it can select an appropriate tuning strategy.

3. *Can we predict the future workload characteristics?* The ability to predict the future workload characteristics is important for autonomic systems because it enables proactive, rather than reactive, resource management. Many researchers have pointed out that proactive, rather than reactive, approach to tuning results in much greater tuning efficiency.
4. *Can we recognize and classify previously unseen, complex, mixed workloads?* Real-world workloads are usually multi-user. Different users can run jobs based on different analytic frameworks. For example, one user could be running Apache Spark jobs, while another user could be running Hadoop MapReduce jobs. Resource orchestration infrastructure, though, will effectively see a hybrid workload that exhibits characteristics of both Apache Spark and Hadoop MapReduce. An autonomic system must be able to adapt to the more complex mixed workload situations.
5. *Can we propose an architecture that can combine the findings from the first 4 items into a coherent autonomic system for big data analytics?* In order to answer the over-arching research question behind this thesis we need to demonstrate that it is possible to combine techniques developed to address the first four questions and modify the big data stack architecture to make it autonomic.

Sections below introduce research investigations intended to find answers to these key research questions.

1.4 Completed Research Investigations

1.4.1 Automatic, On-line Tuning of YARN Container Density

The research presented in Chapter 5 addresses the first of the key research questions. This chapter is based on [13], and presents KERMIT - the first on-line automatic tuning system for YARN. KERMIT optimizes YARN memory and CPU allocations to individual YARN containers, in real time, by analysing the container response-time performance.

Unlike previous automatic tuning methods developed for specific systems, such as Apache Spark or Hadoop, this is the first study that focuses on the more general

case of on-line, real-time tuning of YARN container density and how this affects performance of applications running on YARN. KERMIT employs the same tuning code to automatically tune any system that uses YARN, including both Apache Spark and Hadoop. KERMIT effectiveness was evaluated for Hadoop and Spark jobs using the Terasort, TPCx-HS, and SMB benchmarks. KERMIT was able to achieve an efficiency of more than 92% of the best possible tuning configuration (exhaustive search of the parameter space) and up to 30% faster than basic manual tuning.

This investigation proved that it is possible to tune the big data stack, including the analytic framework and the container orchestration layer, on-line, and produce results that are as good as, or better than, those produced by rule-of-thumb tuning by a human administrator. It was also observed, however, that parameter space searches were expensive. There were situations when excessive searching tended to reduce performance benefits. In some cases, when KERMIT chose to search continuously, performance could actually slow down. Optimising the frequency of parameter space searches was observed to be very important. This trend was also noted by other researchers.

During this investigation the problem was resolved, but it should be noted that the workloads used in the study (benchmarks) were simpler than what a real-world production workload would be. What if the same workload was executed many times? This is a very common situation, because many enterprises run some analytic jobs regularly. Performing a parameter space search in this case would be wasteful. It would be better if the autonomic system could recognize the workload and simply apply a previously used tuning. Conversely, if a brand new workload was detected, it would be better to immediately perform a parameter space search.

The next investigation focuses on being able to identify and classify big data workloads. This is an important capability for future autonomic systems because it enables them to react in a more flexible manner. Depending on the class of the workload being executed, the system may decide to either perform a parameter space search, a look-up of parameter values from history, or a parameter space search that prioritizes certain dimensions of the parameter space.

1.4.2 Machine Learning Based Spark and Hadoop Workload Classification Using Container Performance Patterns

Chapter 6 is based on [12], and addresses the second of the key research questions. This chapter presents a new machine-learning-based technique for identifying and classifying workloads and detecting changes in the workload characteristics.

Big data workload analysis research performed to date has focused mostly on system-level parameters, such as CPU and memory utilization, rather than the higher-level container metrics. Chapter 6 presents the first detailed experimental analysis of container performance metrics in Hadoop and Spark workloads. It demonstrates that big data workloads show unique patterns of container creation, completion, response-time, and relative standard deviation of response-time. Based on these observations, a machine-learning-based workload classifier with a workload classification accuracy of 83% and a workload change detection accuracy of 74% was developed. The observed experimental results are an important step towards developing automatically tuned, fully autonomous cloud infrastructure for big data analytics.

Research presented in Chapter 6 demonstrates that it is possible to use container performance data in combination with machine-learning algorithms to accurately classify big data workloads, and to accurately detect important changes in workload characteristics. It introduces a brand new ensemble-based real-time change detection algorithm for continuous real-time data.

It does not allow us, however, to predict which workloads can be expected in the future. Many researchers working on autonomic computing in the more traditional "small data space" noted that adapting the system pro-actively, rather than reactively, produces much better results. The next investigation aims to address this aspect.

1.4.3 Autonomic Workload Change Classification and Prediction for Big Data Workloads

Chapter 7 is based on [11], and addresses the third of the key research questions. Autonomic systems need to be able to accurately detect important changes in workload characteristics, predict future workload characteristics, and use this information to pro-actively optimise resource allocation and frequency of parameter searches.

Chapter 7 builds on the results achieved and summarized in the previous chapter, and presents the first study focusing on workload change detection, change classification

and workload forecasting in big data workloads. Presented research demonstrates 99% accuracy for workload change detection, 90% accuracy for workload and workload transition classification, and up to 96% accuracy for future workload type prediction on Spark and Hadoop job flows simulated using popular big data benchmarks. The method presented in this chapter differs from prior efforts by other researchers in that it does not rely on past workload history for workload type prediction.

1.4.4 Zero-shot Machine Learning Technique for Classification of Hybrid Classes

Chapter 8 presents the research investigation intended to address the fourth key research question. Real-world workloads are often mixed. Multiple users execute different types of jobs concurrently. Depending on the proportional mix of different jobs, the number of workload classes could become very large despite the fact that the number of analytic frameworks used may not be. For example, users may be running different types of Apache Hadoop and Spark jobs concurrently. Workload characteristics, as observed by the container orchestration infrastructure, may be very different than those observed when only Apache Spark or Hadoop jobs are running.

Machine learning techniques presented in Chapter 6 and Chapter 7 rely on supervised machine learning algorithms. Constructing training data sets for these algorithms is expensive and time-consuming. The research investigation presented in Chapter 8 presents a new Zero-Shot Learning (ZSL) technique intended to dramatically reduce the effort need to construct the training, validation, and test data sets.

Zero-Shot Learning (ZSL) is an advanced machine learning approach that enables the classification of objects without having to explicitly train on examples of those objects. Chapter 4 describes the fundamentals of ZSL.

Chapter 8 demonstrates that multi-user big data workloads can be treated as hybrids of simpler, single-user workload classes, and classified accurately without having to explicitly train on example instances of multi-user workloads. This technique is able to accurately classify both unseen multi-user workloads, and seen single-user workloads using the same classifier. The key results include an 83% classification accuracy for the unseen multi-user workloads, and a 92% classification accuracy for the seen, single-user workload classes.

1.4.5 Autonomic Architecture for Big Data

Chapter 9 is an architecture-focused paper that addresses the fifth key research question. This investigation integrates the results of the previous investigations into a coherent autonomic workload optimization architecture for big data.

KERMIT - the autonomic system presented in Chapter 9 is able to detect changes in workload characteristics, classify and predict workloads, search the parameter space for the optimal configuration parameter values in real time, without human intervention. KERMIT is able to optimize the frequency of parameter space searches by associating search results with the workload class, and caching and re-using them whenever the same workload class is encountered.

KERMIT uses clustering algorithms to identify new workloads and automatically generate labels and training sets for downstream supervised machine learning classifiers. It is also able to detect changes in the workload characteristics, termed workload drift, and compensate for those changes. KERMIT is the first autonomic system able to anticipate the appearance of new, previously unseen, workloads.

1.4.6 Conclusion

The final chapter of this thesis builds on chapter 9 and discusses the implications of the findings presented in the earlier chapters for the architecture and future direction of the big data stack. The findings presented in earlier chapters open the door for several interesting future research directions. This chapter also discusses potential future work.

Chapter 2

Architecture of the Big Data Technology Stack

The main goal of this chapter is to explain the choice of technologies selected as focus areas for this study, and to establish the breadth of applicability of the research presented in subsequent chapters. The focus is not on repeating architectural documentation available in the public domain, but to highlight aspects of the big data technologies relevant to research investigations presented in this thesis.

Sections below provide an architectural overview of the Apache Hadoop stack and the related big data technologies. Relevant aspects of the architectures of the most common analytic frameworks are discussed first, followed by a recap of the orchestration layer architecture, focusing on the resource managers and the containers. The chapter concludes with a discussion of the technological trends and focus areas for this study.

2.1 Architecture Overview

The promise of big data analytics has been driving rapid technology innovation over the past several years. Most big data analytic systems and applications being developed are based on the open-source Apache Hadoop technology ecosystem. Figure 2.1 shows the high-level logical architecture of the Apache Hadoop ecosystem in 2015 [26], while Figure 2.2 shows the same thing in 2019. Although we have seen an explosion in interest and related research into big data, we can see that the architecture has remained broadly very similar, with few notable changes. We will discuss these changes in the paragraphs below.

Figure 2.1: Apache Hadoop ecosystem and logical architecture in 2014.

The big data architecture consists of a number of layers (see Figure 2.1). The base layer is the storage layer. This storage layer must be capable of scaling to many PB of data. Typically, this layer is implemented using the open-source Hadoop Distributed File System (HDFS). Although there have been efforts by commercial vendors to position alternate storage technologies in this space (such as IBM with GPFS), HDFS continues to be the standard due to the low cost (free), reliability, and scalability (see Figure 2.2).

HDFS works by replicating blocks of data among multiple disks mounted on multiple servers. The servers are typically inexpensive, converged architecture, machines capable of hosting many disks within the same chassis. The HDFS behaviour is governed by the configuration settings in the `core-site.xml` and the `hdfs-site.xml` files that are part of the Hadoop configuration. These files contain a number of settings that can significantly influence performance of big data applications. The most important of these are the settings that influence the HDFS block size and the replication factor. These settings, however, can not be tuned dynamically while an analytic application runs because they are determined when the data is originally ingested and stored on HDFS.

The next layer that sits above the storage layer is the resource management layer.

Figure 2.2: Apache Hadoop ecosystem and logical architecture in 2019.

Early in the Hadoop maturity cycle resource management was merged with one of the analytic frameworks (more about these below, see Figure 2.1), but with time it was separated out into it's own layer (see Figure 2.2) whose purpose is to negotiate and orchestrate resources, such as CPU and memory, among different analytic applications.

The Hadoop resource orchestration layer is frequently implemented using Apache Yet Another Resource Negotiator (YARN). Recently, other open-source resource managers, such as Kubernetes and Mesos, have also developed integrations with the big data analytic frameworks such as Apache Spark. We discuss the resource orchestration layer in more detail below, drawing parallels and highlighting differences among these technologies.

Operating on top of the storage and the resource orchestration layers is a growing number of big data analytic frameworks. Initially the Hadoop MapReduce framework played a central role in building analytic applications (see Figure 2.1), and served as the base for other frameworks such as Pig, Hive, and Mahout. These frameworks resolved their operations, such as NoSQL queries, into sequences of Hadoop MapReduce jobs.

As the big data technology stack matured, new analytic frameworks were introduced, and some of the existing frameworks started to re-base on these newer solutions. Good examples of this are recent efforts to re-base Pig, Hive and Mahout on Tez (see Figure 2.2), and the increasingly wide-spread use of Apache Spark. We discuss the

similarities and differences among analytic frameworks in more depth in section [2.2](#).

2.2 Analytic Frameworks

In this section we will focus our discussion on two of the more commonly used big data analytic frameworks: 1) Hadoop MapReduce; 2) Apache Spark. We choose to discuss Hadoop MapReduce because it is at the core of many other technologies, such as the data warehousing framework Hive, the machine learning framework Mahout, the Directed Acyclic Graph (DAG) framework Tez, and many others. Tuning these frameworks, and the applications that are built on top of them, ultimately boils down to tuning Hadoop MapReduce.

Recently, Apache Spark has emerged as an alternative to Hadoop MapReduce. Spark includes APIs that perform functions that are very similar to analytic frameworks of the Hadoop Stack: 1 - Spark core for map-reduce; 2 - DAG API similar to Hadoop Tez; 3 - MLib machine learning API similar to Mahout; 4 - Spark NoSQL API similar to Hive and Pig. Hive, for example, has recently introduced a feature allowing the use of Spark as its core SQL engine. Apache Spark and Hadoop MapReduce are discussed in more detail below.

2.2.1 Hadoop MapReduce

Hadoop MapReduce is the one of the earliest big data analytic frameworks. It uses the map-reduce paradigm to process huge volumes of data in a massively parallel manner. Behaviour of the MapReduce framework is controlled by the configuration settings in the `mapred-site.xml` file that forms part of Hadoop configuration. When performing a computation, MapReduce typically starts by creating a large number of map tasks. These begin execution by loading data from the HDFS. The number of Map tasks is configurable, but usually the default automatic behaviour is used. By default, the number of Map tasks equals the number of the HDFS blocks in the data set, and is thus affected by the HDFS block size selected during data capture and the size of the data set. This can significantly affect the performance of the Map phase, and, thus, the end-to-end performance of the analytic job.

The MapReduce framework interacts with YARN to schedule Map tasks to run on different servers, ideally those servers that contain a copy of the data block required by the task. MapReduce requests an allotment of memory and CPU from YARN. The

amount of memory and CPU that the MapReduce framework requests from YARN is currently configured via the `mapred-site.xml` file. This amount is constant for each task. YARN allocates the CPU and memory to the task using a construct called the *Container*. Physically, each allocated container translates into a JVM started on one of the servers. We discuss this construct in more depth below. Configuration settings that govern the amount of CPU and memory that MapReduce requests from YARN for each task effectively govern how many tasks will be executed in parallel at any given time, and this affects application performance significantly.

After the map tasks complete their work, they write intermediate data to disk, and the MapReduce framework starts a number of Reduce tasks. Reduce tasks begin their processing by fetching the intermediate data not available on the server hosting a particular task from remote servers. This phase of processing is called the shuffle. Once this is completed, the Reduce tasks proceed to combine the output of the Map tasks in the manner desired by the application. The MapReduce framework requests resources from YARN for Reduce tasks in exactly the same way as for Map tasks, and YARN allocates containers to the tasks in exactly the same manner. Unlike for Map tasks, there is no default automatic behaviour for Reduce tasks. The number of Reduce tasks that are started must be configured by the system administrator in the `mapred-site.xml` file. The same is true for the amount of CPU and Memory that MapReduce will request for each task. Both the total number of Reduce tasks, and the number that will run in parallel can significantly affect the performance of the analytic application.

Higher-level analytic frameworks such as Hive and Mahout, decompose SQL queries and machine learning algorithms into a series of MapReduce jobs. These frameworks have their own, additional, configuration settings and configuration files. Changes to these configuration settings indirectly control underlying MapReduce configurations. In all cases interaction with YARN drives the resource allocation to the tasks, and the degree of concurrency and parallelism that will occur when a job is executed. It is important to highlight that MapReduce will request resources from YARN for every Map or Reduce task that needs to be executed, resulting in a *Container* being created for each task.

Figure 2.3: Apache Spark cluster architecture [49]. Figure shared under open-source Apache License 2.0.

2.2.2 Apache Spark

Apache Spark is another popular analytic framework. The Apache Spark distributed architecture is shown in Figure 2.3. Spark can perform very similar types of analytic processing to the Hadoop MapReduce and the related MapReduce-based analytic frameworks. It does so, however, in a somewhat different way.

In Figure 2.3 the role of Cluster Manager can be played by either YARN, Kubernetes, or Mesos (more below). Like MapReduce, the Spark driver program requests containers from the resource manager (Cluster Manager in the figure). Like MapReduce, each container ends up being a JVM running on one of the cluster servers. These JVMs are termed *Executors*. Unlike MapReduce, however, Apache Spark implements a two-level scheduling approach. It re-uses the executors, and schedules multiple tasks to run on them.

Thus, we can see that Hadoop MapReduce and Apache Spark implement different integration strategies, with MapReduce requesting resources (containers) from the resource manager for every task, and relying on the resource manager scheduler to make allocation decisions, and Spark requesting resources from the resource manager less frequently and re-using allocated resources (containers) for running the analytic tasks.

Figure 2.4: Tez integration with YARN.

The majority of analytic applications follow one of these two integration patterns. Figure 2.4 shows the details of Tez-on-YARN integration. Like Spark, Tez re-uses containers, termed, in this case, Tez Task Hosts.

2.3 Resource Managers and Containers

As of writing of this document, almost all big data clusters use one of the three open-source resource managers. We discuss these in more detail in the paragraphs below:

- **YARN.** This is the original integral resource manager of the Apache Hadoop stack. It integrates with virtually all of the big data analytic frameworks and NoSQL databases (MapReduce, Hive, Pig, Mahout, Hbase, Spark, Tez etc.).
- **Kubernetes.** This resource manager was originally developed to orchestrate resources for multi-tier web applications. Recently, integration with Apache Spark has been developed.
- **Mesos.** This resource manager was originally developed to orchestrate jobs running in a data center. It aims to function as an overall distributed operating

Figure 2.5: Apache Hadoop YARN architecture [17]. Figure shared under open-source Apache License 2.0.

system for the data center. It provides an integration with Apache Spark.

The YARN architecture is shown in Figure 5.1. Analytic frameworks, such as the Apache MapReduce and Spark, need to implement a plug-in that wraps an API needed to interact with the YARN resource manager. The client application connects via this plug-in to the resource manager. The resource manager instantiates the application master component, implemented specifically for the client analytic framework. The application master then routes the resource requests to the resource manager. The resource manager uses one of the configured schedulers (not shown in this diagram) to allocate resources. Once the resources are allocated, a *Container* is created to track and manage them. The YARN *Container* is essentially a data structure that keeps track of CPU and memory assigned to a particular task.

The architecture describing Apache Spark integration with Kubernetes is shown in Figure 2.6. Kubernetes has an internal construct called a *Pod*. The Kubernetes *Pod* is similar to the YARN *Container* in that manages the allocation of resources used to execute tasks. Spark executors run in Kubernetes Pods, similar to the way they

Figure 2.6: Apache Spark Integration With Kubernetes [30]. Figure shared under open-source Apache License 2.0.

run in YARN containers. Hadoop MapReduce does not currently have an integration with Kubernetes.

The YARN *Container* and the Kubernetes *Pod* are resource allocation structures used internally by the respective resource managers. By themselves these constructs allow the resource manager to manage allocated resources, but they do not actually enforce these allocations to make sure that, for example, a Spark executor JVM started in a YARN *Container* or a Kubernetes *Pod* exceed the allotment.

Figure 2.7 shows the component flow for Docker. Docker is a container technology that is very often used to provide a physical backing to logical *Container* and *Pod* constructs used by YARN and Kubernetes. YARN, Mesos and Kubernetes provide integrations with Docker, and start docker containers on host servers whenever, for example, a YARN *Container* or a Kubernetes *Pod* is allocated on that host. Docker provides process isolation, and ensures that only the allotted amount of memory, CPU, and disk resources is used.

Figure 2.7: Docker component flow [9]. Figure shared under open-source Apache License 2.0.

2.4 Conclusion

The overview of big data technologies presented in this chapter allows us to draw some conclusions about the current, and the possible future, state of the big data stack, and select the most interesting technologies for deeper research:

- The overall architecture of the big data technology stack, despite intense research and development effort invested over the last several years, has remained consistent.
- There is a clear trend towards 'containerized' applications.
- All popular resource managers have a *Container* construct, and provide integrations with dedicated containerization technologies such as Docker.

- All analytic frameworks need to interact with the resource managers.
- The manner in which the analytic frameworks interact with the resource manager is driven by the analytic framework, rather than by the resource manager, which responds to requests from the analytic frameworks.
- Hadoop MapReduce and Apache Spark are currently the most widely used analytic frameworks, and demonstrate two different integration styles.
- Apache YARN currently integrates with the largest number of analytic frameworks.

Due to these considerations it was decided to focus this thesis investigations on the Hadoop MapReduce and the Apache Spark being used in combination with the Apache YARN.

Chapter 3

Big Data Resource Scheduling

This chapter expands on Chapter 2 and discusses job scheduling in the context of the big data stack architecture. Sections below begin with the discussion on how schedulers and job scheduling fit into the big data stack. Subsequently, this chapter delves into the different types of schedulers and the scheduling algorithms that are predominantly used in the big data space today.

3.1 Resource Managers and Schedulers

All resource managers currently popular in the big data space - YARN, Kubernetes, and Mesos - deploy a pluggable component called the scheduler. The scheduler component is responsible for allocating cluster resources to the different applications running on the cluster. Paragraphs below will examine the YARN schedulers and associated algorithms in more detail. Schedulers provided with Kubernetes and Mesos offer very similar capabilities.

Figure 3.1 shows the architectural relationship between the YARN resource manager, the scheduler, the application master, and the containers. When an application is submitted to the resource manager, it starts the application master. The application master starts the job, analyses its resources needs, and sends the resource request to the resource manager. The resource request asks for a number of containers.

The amount of the CPU, memory, and other resources requested for each container depends on the configuration parameter values entered into the application configuration file. For example, if the application is an Apache Spark job, then the amount of CPUs per container will come from the *spark.executor.cores* property in the Spark

Figure 3.1: The YARN architecture with focus on the scheduler and the application master.

configuration, and the amount of memory will come from the *spark.executor.memory* property.

The resource manager will then pass the request to the scheduler to determine which requests can be met with the resources available on-hand, and which will have to wait. Different schedulers, implementing different resource scheduling algorithms are available. The most frequently used scheduling algorithms are discussed below.

3.1.1 The FIFO Scheduler

The most basic resource scheduling algorithm used in the big data space is the First In First Out (FIFO) algorithm. Figure 3.2 shows graphically how the FIFO resource scheduling algorithm works.

As the name indicates, jobs that are submitted first get executed first. If the application master requests all of the resources in the cluster - any other jobs, that were submitted just after the first job, have to wait until the first job finishes executing and frees up the resources it is using.

The FIFO algorithm is conceptually very simple to understand and does not require any additional configuration. The main drawback is that this algorithm makes no

Figure 3.2: The YARN FIFO scheduler algorithm.

effort to enforce any application performance constraints. If a large job takes all of the resources of the cluster, then all the other jobs, even if they are higher priority, will have to wait until that job finishes and releases resources. The FIFO algorithm is not really suitable for use on shared clusters.

3.1.2 The Capacity Scheduler

The capacity scheduler algorithm addresses the main drawbacks of the FIFO algorithm. It allows applications to share cluster resources by using a logical construct called "job queue".

The capacity scheduler allows users to define a number of job queues via the scheduler configuration file. The user can specify the minimum amount of resources to reserve for a given queue, as well as the maximum amount of resources that the queue can consume should these be available.

This approach allows multiple jobs to execute concurrently, even though the

Figure 3.3: The YARN capacity scheduler algorithm.

response-time for some of the larger jobs may be slower relative to the FIFO model. This algorithm requires additional configuration by the user via an additional configuration file.

3.1.3 The Fair Scheduler

The fair scheduling algorithm is very widely used in the big data space. All of the popular resource managers provide implementations of this algorithm.

The fair scheduling algorithm tries to distribute the resources available on the cluster in such a way as to make sure that each application requesting resources gets a fairly equal share. This is shown in Figure 3.4.

This algorithm also requires additional configuration by the end-user. The administrator defines a queue for each user group, along with a relative weight for this queue. When user matched to queue A submits the first job, it will be allocated all of the resources on the cluster because they are available, and because the algorithm seeks to optimize the job performance. As soon as another user, matched to queue B, submits

Figure 3.4: The YARN fair scheduler algorithm.

job 2 - the fair algorithm will reclaim a portion of the resources originally allocated to job 1. Job 1 will thus slow down, but job 2 will get its fair share of resources and will be able to proceed, in parallel with job 1.

When the user matched with queue B (see Figure 3.4) submits job 3, the fair scheduling algorithm will reclaim some of the resources previously allocated to job 2, leaving resources allocated to job 1 untouched (because it belongs to a different queue). This will be done in such a way as to make sure that job 2 and job 3 get an equal share of resources reserved for queue B. Jobs 1, 2, and 3 will be able to continue in parallel.

The mechanism that the fair scheduling algorithm uses to reclaim resources from jobs that are already executing is called pre-emption. The section below discusses this mechanism in more detail.

3.2 Pre-emption

Pre-emption plays a very important role in ensuring that jobs don't get starved of resources. The system administrator enables pre-emption by changing configuration settings in the resource manager configuration file. One of these configuration settings involves setting a threshold that controls the point at which the resource manager will begin to pre-empt. For example, this threshold could be set to 80%. In this case pre-emption will start when 80% of the cluster resources have been allocated.

To reclaim resources, the resource manager pre-empts already running containers. The pre-empted container process is essentially killed by the resource manager. Any tasks that were running on that container would have to be marked as failed, and would need to be re-started at a later time.

The pre-emption threshold controls how aggressively the resource manager pre-empts. If it is set too low then too many containers will be killed and the overall performance, as measured by the combined job throughput, could suffer. If it is set too high, then some of the jobs could end up starved of resources for a long period of time - again hurting the combined job throughput. Essentially, it is another optimization problem.

3.3 Conclusion

Scheduling algorithms currently popular in the big data space arbitrate resources among multiple concurrently executing applications. They use pre-configured values for each resource allocation, and do not dynamically optimize these values.

The FIFO, capacity, and fair scheduling algorithms are reactive in nature. They re-allocate resources only after receiving a request for resources from the analytic framework. This involves either waiting for the resources to free up, or pre-empting already running containers. Either choice causes performance degradation.

These algorithms do not predict future changes in workload, and they do not try to maintain a pool of contingency resources to meet future demand. Instead, they rely on pre-emption to free up resources if demand increases.

These algorithms are also workload-agnostic. They don't have any knowledge of the executing workload and do not modify their behaviour in any way in response to changes in workload characteristics. Performance optimization has to be achieved through manual configuration by experimentally establishing ideal values for the queue

capacities, weights and the pre-emption threshold. This is very expensive. Due to this most big data jobs are run using sub-optimal configuration settings.

Chapter 4

Machine Learning and Autonomic Computing

In recent years the wide-spread introduction of Machine Learning (ML) algorithms in many diverse domains such as Web-based retail, self-driving cars, medical imaging, fraud detection, stock market prediction, sentiment analysis, natural language processing, machine translation, security, and many others, has revolutionized human-computer interaction. Although many people may not realize this, ML algorithms now have a pervasive influence on our lives. An average person now directly or indirectly invokes them many times during the average day to help with activities like navigating to a desired destination, filtering the content presented in their e-mail reader, selecting products to purchase, and obtaining medical advice.

To be truly autonomic, performance optimization systems need to be able to learn when presented with unforeseen workload conditions. It is easy to see how ML algorithms are a strong fit for this use case. Researchers working on autonomic computing applications realized this early on, and most works in that field leverage various ML algorithms in both similar and different ways [21], [7], [38], [23].

A number of machine learning techniques are used extensively in research presented in this thesis. This chapter presents an overview of the relevant aspects of ML, and discusses how they relate to autonomic computing. Sections below start by introducing ML as a field of computer science, and then proceed to describe concepts, techniques, and algorithms that are the most relevant to the research investigations presented later in this thesis.

4.1 Machine Learning Types and Use Cases

There are many definitions for what the term "machine learning" means. Wikipedia currently defines it like this [54]: "Machine Learning is the study of computer algorithms that improve automatically through time." ML is generally regarded as a subset of Artificial Intelligence (AI). ML algorithms typically include a mathematical model that is capable of extracting new information when presented with a new set of data without requiring that additional code be written by the developer.

ML algorithms can be broadly classified into the following categories:

- **Supervised Learning (SML)**. This type of learning involves presenting an ML algorithm with a fully labelled set of training data that includes examples of inputs, as well as output that the algorithm is expected to produce to match the inputs. During the training phase the algorithm examines the data in the training set, and adjusts certain parameters of its model to produce the desired output. Often, this involves searching the parameter space of the model for optimal values. Thus, a supervised ML algorithm 'learns' the model parameters. After it has been trained using labelled training data, a supervised ML algorithm can be presented with an unlabelled instance, and asked to predict the output. If the output variable is categorical (discrete), then the operation is called classification. If the output variable is numerical (continuous) then the operation is called regression.
- **Unsupervised Learning (UML)**. This type of learning involves presenting an ML algorithm with a set of data that has no labels. Rather than performing classification and regression, the objective of an UML algorithm is to identify patterns in the data. Most commonly this involves either identifying clusters within the data, or anomaly detection.

Sub-sections below summarize popular SML and UML algorithms.

4.1.1 Popular SML Algorithms

SML algorithms have been especially successful in many industry domains. Examples of SML algorithms include (not a complete list, but some examples relevant to later chapters of this thesis):

- **K-Nearest-Neighbour (kNN).** The kNN algorithm can perform both classification and regression. When presented with a data point, the algorithm examines k nearest neighbouring data points. The data points are typically embedded in the multi-dimensional real-number feature space $\mathbb{R}^D$. Euclidean distance is typically used to establish the distance between points. If the classifier is performing a classification, then the new data point will be assigned a class based on the majority vote of the k nearest neighbours. In some cases, the votes can be weighted to give greater weight to the closest points over more distant ones. If the classifier is performing regression, then the numeric value of the new data point will be the average, or weighted average of k nearest neighbours. The kNN classifier continues to be used in many applications, such as facial recognition. This algorithm is prone to a problem often referred to as the curse of dimensionality. The curse of dimensionality refers to situations whereby most points tend to be very far apart and mostly equidistant in the high-dimensional feature spaces. The value k is not learned. It is usually established heuristically, and is referred to as a hyper-parameter. Hyper-parameters are tuning parameters that have to be set by a human administrator.
- **Linear Regression (LR).** The linear regression algorithm is another relatively simple algorithm that continues to be frequently used in modern applications despite its limitations. As with kNN, LR can be used for both classification and regression tasks. The training phase of the algorithm consists of fitting a linear hyperplane to the data by minimizing the Residual Sum of Square (RSS) distances from each labelled point in the training set to the hyperplane. A binary classifier can be constructed by considering all points to one side of the hyper-plane as one class, and to the other side - as the other class. The LR algorithm is frequently used for regression tasks. Most data, even highly non-linear overall, can be approximated as linear over a limited range of feature values. The LR algorithm can then be used to predict values for continuous numeric features.
- **Logistic Regression (LgR).** The LgR algorithm is used to determine whether a particular condition is true or false. For example, the LgR algorithm can be used to test whether an image contains a particular object, such as a dog or a cat. The LgR algorithm uses the logistic function, which converts log odds to

probability, to determine whether the dependent variable fits a specific category based on the values from one or more other variables. The logistic function is sigmoid-shaped. It takes any real value and outputs a value between 0 and 1. The LgR algorithm is trained using maximum likelihood estimation. This is an iterative process during which the sigmoid-shaped logistic function probability curve is shifted around to maximize the likelihood for the training data.

- **Support Vector Machines (SVM).** The Support Vector Classifier (SVC) uses a linear decision boundary to solve classification problems. In multiple dimensions, this boundary is a hyper-plane that separates two classes of data points. Data points that are closest to the hyperplane, and influence its location and orientation are called support vectors. The distance from the closest data point to the boundary is called the margin. In those cases where the two classes of data points are clearly separable the margin is called a hard margin. In those cases where they are not - the margin is called a soft margin. Training an SVC involves finding the optimal values for the slope and intercept vectors for the hyperplane. The hinge loss function is used during the training process to optimize the values for these learned parameters. In many cases it is not possible to separate the data using a linear boundary. In this case data can be projected into a higher-dimensional space, by introducing new dimensions derived from original ones, where a clear separation may be possible. This technique is called the kernel trick. The SVM algorithm uses the kernel trick and the cross-validation technique (described in more detail below) to find an optimal SVC for the training data. The SVM algorithm has been used extensively for applications such as text and hypertext categorization, image classification, character recognition and protein classification.
- **Decision Tree (DT).** The DT algorithm can be used for both classification and regression. This algorithm works by maximizing information gain - a measure of how much order has been introduced into the data set by performing a data split. The DT algorithm examines each feature in the feature vector, and determines how to best split the data using this feature to maximize information gain for the data. The feature which produces the most information gain when the data set is split on it, becomes the root of the tree. This process is then repeated for the child nodes, examining information gain at each node, until all the features

have been used and the features have been organized into a tree structure. The DT algorithm is easy to express in terms of human-understandable rules. This gives the results produced by the DT algorithm high explainability. The DT algorithm is often used for this reason. Decisions trees, and the concept of information gain, are discussed in more detail below.

- **Random Forest (RF).** The RF algorithm is an example of an ensemble algorithm. Ensemble algorithms use a collection of weak learners to determine the answer. In this case the weak learners are individual decision trees that combine to form the forest. The root feature, and subsequent branches, of each decision tree are selected pseudo-randomly rather than by selecting the feature that maximizes the information gain. Each tree is trained on a sub-set of the data set. The majority vote among among individual decision trees is taken to establish the classification output. The average of weak learner outputs is used for regression. Like the DT algorithm, the RF algorithm has very good explain-ability. The RF algorithm is described in greater detail below.
- **Artificial Neural Networks (ANN).** The ANN algorithm is designed to simulate biological processes that take place in the human brain. The basic building block is the artificial neuron. The artificial neuron is a mathematical function designed to simulate the behaviour of the actual biological neuron. The artificial neuron takes a number of inputs, multiplies each input by a corresponding weight and adds a corresponding bias. The resulting numbers are fed into an activation function, which then produces the output of the neuron. Constructing an ANN model generally involves defining a large number of artificial neurons organised into layers. The model must have at least two layers - the input layer and the output layer, although commonly a number of intermediate layers are also defined. Each layer can have a different activation function. Some popular choices for the activation functions are the ReLU function and the Sigmiod function. The weights and balances used by each artificial neuron need to be learned during the training process. The training process involves presenting the ANN algorithm with a labelled training data set.
- **Convolutional Neural Networks (CNN).** The CNN algorithm differs from the ANN algorithm in that a CNN model contains convolutional layers, usually in addition to the directly connected layers of artificial neurons that are also

used by the ANN. Instead of a vector, a convolutional layer takes a tensor as input. A tensor is essentially a multi-dimensional array. In the case of images it's dimensions are usually: a) height (in pixels), width (in pixels), and depth (number of channels - for example Red Green and Blue). The convolution operation involves moving a filter (also called a kernel) over the layer in steps. The filter is also a tensor with width, height and channels. The number of filter channels must match the number of input channels. At each step, for each channel, a dot product between the filter matrix and the matching segment of the input layer matrix is calculated. The result, usually a real number, is fed into an activation function. The most commonly used activation function is the Rectified Linear Unit function (ReLU). The ReLU function outputs 0 if the dot product is negative, and the value of the dot product if it is positive. The output of the activation function becomes the output value for that pixel layer. The filter width, height and number of input channels are hyper-parameters that must be configured by the developer. The contents of each filter are learned during the training stage using back-propagation similar to the way weights and balances are learned for the ANN algorithm. CNNs usually include multiple convolutional layers interspersed with pooling layers. The output of these layers is usually flattened into a vector and fed into a directly connected layer(s) for the final classification. CNNs are used predominantly, though not exclusively, for image classification.

- **Recurrent Neural Networks (RNN).** An RNN is a type of ANN that contains loops. Loops allow the RNN to store state. This makes RNNs very useful for working with time-sequences of events. RNNs are frequently used in applications such as natural language processing and machine translation. The key concepts that underpin the RNN algorithm are discussed in more detail below.
- **Long Short-Term Memory Neural Networks (LSTM).** The LSTM algorithm is a type of RNN that contains special units - such as input and forget gates. These units enable better tracking of long-range dependencies. The LSTM algorithm is discussed in greater detail below.

4.1.2 Popular UML Algorithms

The UML algorithms do not require training. They are used extensively for data exploration, and often as part of ML pipelines (discussed further below) in combination with the SML algorithms. Two of the most common techniques are clustering and anomaly detection.

Clustering techniques are used to discover the presence of different groupings in the data. This approach is relevant to this thesis because it can be used to automate discovery of the different workload classes within the data. Two clustering algorithms have been used extensively by researchers working on autonomic computing problems, and are evaluated in the subsequent thesis chapters:

- **K-Means.** The K-Means algorithm begins by randomly generating K cluster centroids. The number K is an integer hyper-parameter that must be known before-hand, and must be manually configured by the human user. K-Means then calculates the Euclidean distance from each data point to each centroid. It then performs the following steps:

1. Assign each data point to the closest centroid using Euclidean distance.
2. Re-calculate the coordinates of the centroids using the only data points assigned to that centroid.

These steps are repeated in cycles until there are not further changes in the centroid coordinates. All data points have essentially been classified into K classes without the need for explicit labelling by a human practitioner.

- **DBSCAN.** Unlike K-Means, the DBSCAN algorithm does not require that the number of clusters be known before hand. The acronym DBSCAN stands for Density-Based Spatial Clustering of Applications with Noise. The algorithm defines two hyper-parameters:

1. **minPts.** This hyper-parameter is often represented by the symbol μ . It determines the number of points that need to be close together in order to be considered part of the same cluster.
2. **eps.** This hyper-parameter is often represented by the symbol ϵ . It determines the distance that will be used to locate additional points in the neighbourhood of any data points.

The DBSCAN algorithm performs the following steps:

1. Pick a point in the data set at random.
2. If there are μ points within ϵ of this data points then these points are part of the same cluster.
3. Repeat the step above for all neighbouring points recursively.
4. Repeat the first step, and subsequent steps until all points in the data set have been visited.

At the conclusion of the algorithm all data points are grouped into density-based clusters.

Both K-Means and DBSCAN can be used for workload class discovery. The main disadvantage of the K-Means algorithm is that it requires the number of clusters K to be known before-hand. Often, though, this is what we want to find out. Obviously, if this number is entered by a human practitioner, it can introduce bias into the final result. This problem can be resolved by using a technique called the elbow method.

The elbow method involves running the K-Means algorithm on the data multiple times, with increasing values of K , and plotting K vs. the mean distance between data points and their centroids. Once the resulting curve is observed to flatten - the value of K at the 'elbow' of the curve can be selected.

Another important difference between DBSCAN and K-Means is the underlying premise of the clustering mechanism itself. Figure 4.1 shows a two dimensional comparison of several possible data point arrangements, and how they would be grouped by DBSCAN and K-Means. DBSCAN performs better than K-Means at clustering points when they form geometrically complex distributions in the feature space. This is due to the fact that K-Means relies on the distance to the centroids as the key clustering criteria, while DBSCAN relies on density. Points arranged in concentric circles, for example (see Figure 4.1), would not be correctly grouped (classified) by K-Means. Points in a homogeneous region would be falsely grouped into several clusters by K-Means, if K was not equal to 1 (see Figure 4.1).

Sections below expand further on several supervised ML algorithms, and ML aspects used in the research investigations presented in the subsequent chapters.

Figure 4.1: K-Means vs. DBSCAN clustering examples. Image from [46] is shared under open-source MIT license.

4.2 Statistical Classifiers

Statistical classification is a sub-field of supervised machine learning. Statistical classifiers aim to predict the class of an instance given a training set of instances. The statistical classifier will pick the category for the instance by examining its feature vector and comparing it to the feature vectors of training samples. The instance being classified will be assigned to the category whose known instances have the most statistically similar feature vectors.

Training a statistical classifier involves calculating the key statistical measures, such as the mean and the standard deviation, for the features in the feature vector in each category. As such, the model of a statistical classifier can be readily incrementally updated as new training data becomes available. Due to this statistical classifiers are often used for on-line, near-real-time classification and learning scenarios.

Naive Bayes classifiers make a very important assumption. They assume that the features used to calculate the class posterior probability are independent. In most cases this assumption is not strictly true, but even in those cases these types of classifiers work surprisingly well. This can certainly be said to be true for workload data, where features such as CPU and memory utilization are not independent.

The *ChangeDetector* component presented and discussed in chapters 6 and 7 is an example of a statistical classifier. This classifier does not require explicit training. It works on-line, classifying analytic window data as changed or not changed relative to the previous analytic window. This classifier uses the Welch’s statistical significance test to compare populations of data points observed by the analytic windows. The Welch’s statistical significance test assumes that the features are independent and

Figure 4.2: The decision tree logical structure.

normally distributed. Although, as mentioned above, this is not strictly true for most of the workload features - this approach was observed to work very well.

4.3 DecisionTree and RandomForest Algorithms

The decision tree algorithm is widely used in machine learning applications because it has been proven effective in many applications and because it can be readily understood by human practitioners. Decision trees can be used for both classification and regression. They can work with a combination of continuous, categorical and ordinal data. In this thesis decision trees are used for classification.

Figure 4.2 shows the logical structure of a decision tree. The decision tree consists of two types of nodes - decision nodes and leaf nodes. The high-level algorithm for training the decision tree has the following steps:

1. Create the root node for the tree, which contains the entire set of data points.
2. Select the best feature in the feature vector to split the data set on. This is done using suitable selection measure. Usually the selection measure used is either information gain or the Gini index.

3. Split the data set in of the root node using the feature that maximizes the selection measure. This results in two sub-trees. Numeric, continuous feature data, such as the workload feature vector data, is first sorted from lowest to highest value, and the threshold is gradually moved from the lowest to highest value to maximize the selection measure of the split.
4. Repeat this process recursively for each decision node until it is no longer possible to split the nodes because they contain only one class of data point. These nodes are the leaf nodes.

To classify a data point at run-time the algorithm examines the feature vector of the data point and simply walks the tree starting with the root node applying the learned test for each split until it hits a leaf node. Decisions made by the algorithm in the process can be easily understood by a human. For example, the workload was classified as the Apache Hadoop map processing because:

1. CPU utilization > 90
2. Disk read > 1000
3. Disk write < 1000
4. Network read < 1000
5. Network write < 1000

In contrast to the model elements of some of the other machine learning algorithms, such as the weights and biases of ANNs, these generated rules are easy to understand.

Decision trees can achieve great accuracy, are easy to understand, and can approximate very complex functions. They are also prone to over-fitting.

The random forest algorithm is an ensemble algorithm. Ensemble algorithms use a collection of weak learners rather than one strong learner. Often, this improves the accuracy on the test set relative to using a single strong learner.

The random forest algorithm uses many trees. Each tree in the random forest is trained using a subset of training data points selected with replacement from the full set. This is known as bagging. Each tree is a full scale decision tree. The difference from the basic decision tree algorithm is that for each candidate split for each tree a

subset of features is chosen at random, rather than using the full set. This is known as feature bagging.

To classify a data point, each tree in the forest follows the generated splits (which are different for each tree) using the feature vector data of the data point, and predicts the class label. The label predicted by each tree is called a vote. Once all of the votes are ready, the random forest algorithm counts the votes. The class with the most votes wins, and is returned by the algorithm.

Bagging techniques used by the random forest algorithm allow it to reach the same, or better, accuracy than the base decision tree algorithm, but also make it less prone to over-fitting.

The random forest algorithm is used as the base for the *WorkloadClassifier* and *TransitionClassifier* components described in chapters 6 and 7.

4.4 Artificial Neural Networks

Artificial neural networks are also called feed-forward neural networks, or Multi-Layer Perceptron (MLP). The basic architecture of an ANN is shown in Figure 4.3. Although at a minimum the ANN consists of an input layer and an output layer, it usually will contain one or more hidden layers. Deep ANN could contain dozens to hundreds of hidden layers. Each layer contains a number of artificial neurons. The key concepts behind the ANN architecture and the training process are discussed in the sub-sections below.

4.4.1 ANN Architecture

The input layer of an ANN consumes the input vector, and feeds the values forward to the next layer. Its role is strictly to read in the input vector data. The input vector can have a very large number of columns. For example, in Natural Language Processing (NLP) applications, where a dictionary with one-hot encoding is used, the input vector can have thousands or even tens of thousands of rows.

Figure 4.3 shows how ANNs work. Each neuron is connected to every neuron in the previous layer. It has multiple inputs, and a single output. Input layer neurons only have a single input, as well as output. The role of the input layer is to simply store the values of the input vector, and feed them forward to one of the hidden layers.

Figure 4.3: Architecture of a simple artificial neural network.

Each neuron of the hidden layer is connected to every output of the input layer. Figure 4.4 shows the architecture of an artificial neuron. For every neuron of the hidden layer, each input value (output value of the input layer) is multiplied by a variable called a weight (ω_i). The resulting products are summed. Changes to different weights have the effect of emphasizing certain connections between neurons - assisting with learning important patterns. This bears similarity to how biological neural networks work.

After the products of input values and weights are summed, another variable called a bias (β) is added to the sum. The bias also helps learn different patterns of data, but in a way that is different from the effect brought on by changing the weights. The bias term can help fight overfitting during the ANN training process. This difference in impact is discussed in more detail below as part of the discussion focusing on the role of the activation function.

Once the products of the weights and the input values, and the bias, have been added together, the resulting value is fed into an activation function. The activation function produces the output value for the neuron. This output value is called the activation, and it becomes the input value for neurons that comprise the next layer of the ANN. The activation of the neuron shown in the Figure 4.4 can be summarized by the following equation:

Figure 4.4: Architecture of the artificial neuron.

$$a_i^l = f\left(\sum_{j=1}^m \omega_j a_j^{l-1} + \beta_i\right)$$

Here a_i^l is the activation value of neuron i of layer l , m is the number of neurons in the previous layer, and j is the index of a given neuron in the previous layer $l - 1$.

Activation functions can vary by layer. Different activation functions have different strong points and drawbacks. Some of the most popular activation functions are:

- **Sigmoid/Logistic.** This function squishes the value of the input to an output between 0 and 1. The output value can be interpreted as the probability that the output belongs to a certain class, and is often used in the output layer of the ANN. This function suffers from the 'vanishing gradient' problem. When the input values are small or large the slope of the sigmoid function is very small, making it difficult to find optimal values using gradient descent (discussed further in sub-section 4.4.2).
- **tanh.** The shape of the tanh - the hyperbolic tangent function - is similar to the sigmoid function. This function, however, returns an activation value between -1 and 1. Unlike the sigmoid function, the outputs are zero-centred. This makes it easier to model situations where inputs can have strongly negative, positive, or

neutral values. The main disadvantage is the same as for the sigmoid function. The tanh and the sigmoid functions are computationally intensive compared to some of the other options.

- **Rectified Linear Unit (ReLU).** This function outputs 0 if the value of the input is negative, and the value of the input if it is greater or equal to 0. The main disadvantage of ReLU is that when inputs approach 0, or are negative, the gradient of this function becomes 0. This makes it impossible to perform backpropagation and learn.
- **Leaky ReLU.** This version of the ReLU function avoids the gradient problem discussed above for the ReLU activation function by introducing a small positive slope in the negative area of the graph. The trade-off is that now the results produced by the function are not consistent for negative input values.
- **Binary step function.** This function returns 1 if the input is above a certain threshold, and 0 otherwise. Like the sigmoid function, this activation function can be used in the output layer. Its binary nature, however, precludes it from being used to construct multi-class classifiers.
- **Softmax.** The softmax function is used mainly in the output layer of the ANN. It takes an input vector of logits and converts them into a vector of probabilities that sum up to 1. Its main advantage is that it is able to classify inputs into multiple categories.
- **Linear/Identity.** This activation function produces an output proportional to the input following the formula $y = cx$, where A stands for the activation and c is a slope constant. It is not possible to use backpropagation to train an ANN when using this function, because the derivative of a linear activation is a constant (see backpropagation discussion in section 4.4.2). The ANN becomes essentially equivalent to the linear regression model.

The output layer of the ANN produces the desired output. Figure 4.3 shows a simple binary classifier ANN. In this case, the input vector holds 5 real numbers. The output layer uses the sigmoid activation function, and produces a single real number between 0 and 1. If the output is rounded to the nearest integer then the result of 0 indicates class 1, and if it equals 1 then class 2.

All of the key architectural elements of ANNs have now been discussed. ANNs are supervised learning algorithms that need to be trained on labelled examples. The training process for an ANN is discussed below.

4.4.2 Training ANN

Modern ANNs are usually trained by initializing the weights and biases to random values, and then using the backpropagation algorithm to find the ideal combination of weights and balances to minimize the loss function. The concept of applying backpropagation of errors to train ANNs was first described by Rumelhart et al. [47]. In order to understand how backpropagation works, it is important to understand the following:

- **The loss function.** The loss function evaluates the difference between the prediction produced by the ANN and the training examples. The loss function most commonly used for ANNs is simply the Mean Square Error (MSE) loss [55]. Another frequently used loss function is the cross-entropy loss.
- **The Gradient Descent (GD) search algorithm.** This classic search algorithm is commonly used in optimization problems. The GD algorithm explores the parameter search space by incrementally moving in the direction of the steepest gradient. The GD algorithm is not guaranteed to find the global minimum within the search space, and will frequently get trapped in a local minimum. In many situations this is regarded as being good enough. There are more advanced variants of the GD algorithm. Some of these are discussed in the paragraphs below.
- **The chain rule.** The calculus chain rule is used to derive the partial derivatives of the loss function, and calculate the gradients needed by the GD algorithm. Derivatives of all the activation functions used in an ANN must be known at design time.

The core steps of the back-propagation algorithm are:

- When presented with a training example, calculate the loss at the output layer. If MSE loss is used, then differences between the output layer neuron activations and expected training values are used.

- Use the chain rule and partial derivatives for all weights and biases to calculate gradients associated with each of these variables.
- Calculate the step size that will be used for the next GD iteration by multiplying the gradient values by the *learning rate* hyper-parameter.
- Once the step sizes for the outer layer have been calculated and applied, the differences between the previous, usually hidden, layer activations and the back-propagated training example values can be calculated.
- Repeat the procedure for the previous hidden layer, until goals for all hidden layers have been established and the gradients associated with all of the weights and the biases have been computed.
- Repeat the entire procedure for all training examples, and average the step sizes for each weight and bias over the entire training data set.
- Apply the steps to all the weights and the biases as part of the GD algorithm.
- Repeat the entire process until the GD minimum has been found.

Treating the weights collectively as a vector, the gradient of the loss function with respect to ω can be expressed as a vector of partial derivatives:

$$\frac{\partial L}{\partial \omega} = [\frac{\partial L}{\partial \omega_1}, \frac{\partial L}{\partial \omega_2}, \dots, \frac{\partial L}{\partial \omega_z}]$$

Here L is the loss function and z is the total number of weights used in the ANN that is being trained. For a single weight ω_j^l the partial derivative can be written out as follows:

$$\frac{\partial L}{\partial \omega_j^l} = \frac{\partial \sum_{j=1}^m (a_j^l - \hat{y})^2}{\partial \omega_j^l}$$

In the case shown in Figure 4.3 the value produced by the sigmoid activation function of the single output layer neuron would be subtracted from the expected value of the labelled training sample. The difference would be squared. An average for all of the differences, for all training samples, would then be computed. If the network contained multiple neurons in the output layer, then the differences for all the output values would be computed, and included in the averaging process, as well.

The algorithm works by first examining the impact of weights and balances within each neuron of the output layer on the final output activation. The algorithm computes the squared error between each output layer neuron activation value, and the training sample value. Based on this difference, for each neuron in the output layer, the algorithm decides how much to nudge the weights and bias that neuron. The amount of change to be applied to the weights is proportional to the activation values of the neurons in the previous layer. Weights associated with larger activation value are nudged more than those associated with lower activation values. This process is repeated for all the neurons in the output layer. Once this is accomplished, the process is repeated for all samples in the training set, and the average desired change for each weight and bias in the layer is calculated.

The back-propagation algorithm then decides by how much the activation values of the neurons in the previous hidden layer should be changed. With the goals for the next layer set, the process is repeated for the next to last layer, and then for each subsequent layer back.

One of the variants of the gradient descent algorithm is the Stochastic Gradient Descent (SGD) algorithm. The SGD algorithm, like the basic variant, moves in the direction of the steepest gradient between iterations, but periodically switches direction at random. This help SGD avoid getting trapped in a local minimum.

4.5 Recurrent Neural Networks and Long Short-Term Memory Networks

The basic ANN architecture can serve as a building block for a more complicated Recurrent Neural Network (RNN). The ANN, when presented with examples of objects, learns the weight and the bias values that will predict the correct label. This approach, however, does not include any notion of time or sequential order among objects.

The RNN extends the ANN architecture to introduce the notion of order dependency. Figure 4.5 shows the basic architecture of a Recurrent Neural Network (RNN). The main difference is that at any given time t the input to the RNN $X^{<t>}$ is combined with the activations produced at a previous time $a^{<t-1>}$ using the tanh activation function to produce the output $Y_{<t>}$.

The activations computed at time $t - 1$ give the RNN unit a memory of what

Figure 4.5: RNN architecture.

happened during the previous step. RNNs can remember events that occurred many steps back thanks to this mechanism. In fact, there are no limitations on how far back an RNN can remember. Due to this RNNs are most commonly used for natural language processing applications.

RNNs are trained using the algorithm called Back-Propagation Through Time (BPTT). This algorithm has all the steps described above for the regular back-propagation algorithm, but adds gradients between different time positions. This involves multiplication of the gradient of the layer at time t with that at layer $t - 1$. If the L_2 norm of this gradient is less than 1 then repeated multiplication during BPTT results in the gradient quickly tending to zero. This is known as the vanishing gradient problem, and it effectively prevents the RNN from learning long period dependencies.

If the L_2 norm of this gradient is greater than 1 then the gradient goes to infinity exponentially fast and causes the BPTT to fail by returning the model loss as NaN. This is known as the exploding gradient problem. The Long Short-Term Memory (LSTM) neural network architecture aims to fix these problems.

Figure 4.6 shows the architecture of an LSTM block. The vanilla LSTM architecture includes, in addition to the input and output gates, the block input and forget gates. These additional layers allow the LSTM cell to maintain state within the cell. The forget gate limits the depth of dependencies that need to be remembered and thereby mitigates the exploding and vanishing gradient problems.

LSTM neural networks are widely used and very successful in natural language processing applications. The *WorkloadPredictor* component presented in chapter 7

Figure 4.6: LSTM architecture. This figure is Figure 1 from [16].

uses the LSTM algorithm to predict which workload class could be expected in the near-term future.

4.6 Applying Machine Learning Techniques - Methodology

As ML application development projects became more and more common it became possible to describe a general best practices approach. This approach generally involves the following steps:

1. Data acquisition.
2. Data cleansing and transformation.
3. Algorithm selection.
4. Model selection.

ML algorithms learn from data, and, thus, it is logical that the first step in any ML project is to get access to the right type and volume of the data. Securing this access is often not straight-forward. In the case of workload data - these can be obtained by either capturing data from a live production environment, or by executing

benchmark applications, designed to simulate what happens in production, in a testing environment and extrapolating to what would happen in production.

It is important to point out that the overall approach is often iterative in nature, with several, or even all, of the above steps having to be repeated several times before satisfactory results are achieved.

4.7 Model Selection and Evaluation

Once the most appropriate ML algorithm has been selected, the next step is to fit the right model to the training data. For example, for the ANN algorithm this involves choosing the number and size of the directly connected layers of artificial neurons. For the CNN algorithm this involves selecting the number of convolutional, pooling, and directly connected layers, the size of the filter (kernel), and the amount of padding. Many models, such as the LR models, include a regularization hyper-parameter λ that helps with the bias-variance trade-off.

The bias-variance trade-off is of critical importance for any ML project or application. Since this aspect is referred to in several of the investigation that form part of this thesis, this section expands on this very important concept.

Evaluation of an ML algorithm performance involves assembling the following sets of data:

1. **Training set.** As the name implies, this set of data is used to train the supervised ML algorithm. The algorithm model is trained using this set and the loss function, until the loss has been minimized.
2. **Validation set.** This data set is used to evaluate different hyper-parameter settings. For best results the algorithm being trained should not see data from this set while the model parameters are being learned.
3. **Test set.** This set is used to evaluate the expected performance metrics of the model. The test set should mimic the real production data as closely as possible. As with the validation set, the algorithm that is being trained should not see any of the data in this set while model parameters are being learned. This set should also not be used to tune the hyper-parameters to avoid introducing bias into the final results.

Figure 4.7: Bias-variance trade-off and model complexity illustrated.

The term *bias* refers to systematic error introduced when training a SML algorithm on the training set. Expressed mathematically, the bias is an additive term in the model equation. An algorithm with high bias will display a relatively high prediction error on the training set, but may display greater accuracy on the test set. This relationship is illustrated in the Figure 4.7.

The term *variance* refers to random error, or noise, present in model. A given model may have low bias and high variance. In this case a very low prediction error on the training set, but high prediction error on the test set may be observed. When this occurs, the model is said to *overfit* the data.

An alternate situation may arise when the model has high bias and low variance. In this case a high prediction error on both the training set and the test set may be observed. When this occurs, the model is said to *underfit* the data.

In practice it is often difficult to find a model that will neither underfit nor overfit the data. The goal is to optimize the model performance by reducing the prediction error on the test set. To accomplish this it is often necessary to add a regularization term to the model which effectively introduces a penalty. This term serves to increase

the bias, thereby increasing the prediction error on the training set, but reducing it on the test set. The process of finding this optimum is the bias-variance trade-off. In Figure 4.7 this optimum is the prediction error minimum of the test sample curve.

The term *model complexity* refers to the number of variable parameters that a given model has. The variable parameters could be those that are learned, as well as hyper-parameters that are configured by the data scientist. The more variable parameters a model has - the more complex it is, and the more likely it is to overfit. Alternatively, a model with few variable parameters is likely to underfit.

The key goal of the model selection process is to find the right level of complexity for the model, that will avoid overfitting and underfitting, and produce the minimum prediction error on the training set. For the ANN algorithm, for example, this involves selecting the number and size of directly connected layers, thereby changing the number of weight and bias variables that are learned during the training process.

4.8 Advanced Machine Learning

Although supervised machine learning algorithms have demonstrated great success in many different problem domains and applications, they do have one key weakness - the cost and effort associated with assembling the data sets needed for training. Accuracy, and other performance measures, of a supervised learning algorithm will be strongly affected by the quality of the data sets used for training, validation and testing. This is where the difficult part comes in - assembling a good data set is often time-consuming and expensive.

The real world contains a huge number of object classes. This is certainly true for workloads. The key workload characteristics are determined by many different factors - the size of the data set, the number of the users, the type of queries being executed, and the analytic framework being used. Different combinations and permutations of these factors can produce very different workload characteristics, and hence different workload classes, even within a single domain such as big data analytics. Multiple examples of each class are needed to train a supervised learning algorithm effectively, and it is easy to see how the cost of obtaining and labelling these samples can be significant.

Transfer learning is a relatively new salient within the broader machine learning field. It refers to the potential of training an algorithm on one data set/domain and

then be able to perform classification and regression operations in a different domain. Transfer learning offers the promise of being able to reduce the cost of assembling the data sets needed for training.

The field of transfer learning has generated a great deal of interest among researchers, and this has lead to the development of several different transfer learning techniques:

- **Zero-Shot Learning (ZSL).** The ZSL approach does not require any actual examples of a class that is to be learned. Instead the user must provide the semantic information that allows the algorithm to construct the semantic space and the class prototype that will be used to classify an object when presented with an actual instance.
- **One-Shot Learning (OSL).** In OSL, a single instance of a class is used in conjunction with either model parameters, features, or contextual information from previously learned classes to learn the new class.
- **Few-Shot Learning (FSL).** FSL aims to predict the correct label for a class based on a small number of training examples.

ZSL is particularly interesting from the workload classification perspective. Most real-world workloads tend to be multi-user workloads. Different users could be executing different types of jobs concurrently. The resulting feature vector could be essentially a mixture, or a hybrid, of feature vectors associated with each type of job individually.

Figure 4.8 demonstrates how a ZSL classifier can work. In this example the previously unseen images of animals can be classified using descriptions of attributes that comprise the image.

For example, instead of learning what a zebra looks like, the classifier learns the key attributes that are always found in the image of a zebra - black colour, white colour, stripes - and those that are not. The classifier is trained to learn the attributes and is then able to identify a previously unseen image of a zebra.

ZSL is a core part of the research investigation presented in Chapter 8. That chapter presents a new ZSL technique that can simplify the identification of workload attributes. The technique is used to build a classifier capable of identifying multi-user workloads without having to explicitly train on examples of these workloads.

Figure 4.8: A ZSL example - animals with attributes. This graphic is Figure 1 from the research paper by Lampert et al. [31].

4.9 Combining Machine Learning Techniques

All machine learning algorithms have inherent limitations. It is often possible, however, to work around these limitations by chaining the different algorithms together in an architectural construct often called a pipeline. The pipeline pattern can be used for both off-line model training, and for on-line predictions.

For example, predicting future values in a multi-variate time series of continuous real number data can be difficult. Especially so if the data can show abrupt, highly non-linear, changes. In this case regression techniques, such as linear regression, may not work well. If the data is first transformed from continuous real-number data to categorical data, then the problem becomes easier to solve because a more suitable machine learning algorithm capable of predicting abrupt changes can be applied.

Chapter 7 describes one such pattern. In this case the multi-variate, continuous real-time data are fed into a random forest-based classifier that classifies the data into one of several pre-determined categories. This happens in real-time. The stream of multi-variate, continuous, real number data is thus transformed into a stream of labels. The LSTM algorithm, described above, can now be used to predict which labels are likely to occur next.

Chapter 9 describes an off-line machine learning pipeline. In this case an unsupervised clustering algorithm DBSCAN serves as the initial stage of the pattern. Once the different classes have been discovered by DBSCAN, the training data for the supervised classifiers can be generated, and the classifiers can be trained using these data.

4.10 Conclusion

The field of machine learning has revolutionized software engineering during the past decade. There are many unsupervised machine learning algorithms that can be used to discover patterns in data. There are also many effective supervised machine learning algorithms that can be used to accurately classify data and perform regression. Advanced machine learning techniques, such as ZSL, can be used to train the system to recognize and classify previously unseen classes. This allows us to significantly reduce the training effort for the system and give it the capability to anticipate the appearance of new classes.

Machine learning algorithms can be combined using the pipeline architectural pattern. Combining different algorithms enables more accurate classification, regression, and prediction. Chapters 6, 7, and 9 describe examples of machine learning pipelines that effectively solve classification and prediction problems.

Chapter 5

Automatic, On-line Tuning of YARN Container Density

Research presented in this chapter focuses on finding an answer to the first research question presented in Chapter 1: "Can we automatically tune key configuration parameters for big data frameworks on-line?". Findings discussed in the sections below review previous work on automatic tuning of big data technologies, discuss the relevant aspects of the YARN architecture, and demonstrate that it is indeed possible to automatically tune key YARN configuration parameters on-line. This chapter is based on [13].

5.1 Problem

There have been a number of attempts to automatically tune big data applications [19], [18], [4], [3], but they focused specifically on tuning the Apache Hadoop MapReduce framework only. This is the first study that focuses on the more general case of online, real-time tuning of YARN container density and how this affects the performance of applications running on YARN. The term "container density" refers to the number of containers started by YARN at a given point in time to run in parallel on the cluster. Running too few containers may leave the cluster under-utilized, resulting in sub-optimal job performance. Running too many containers at the same time can create contention for key resources such as CPU and memory, again leading to poor application performance.

5.2 Contribution

As part of our study we developed an on-line automatic tuning Java library, called KERMIT, designed specifically for on-line automatic tuning. Although in this study KERMIT was integrated with YARN, it exposes an integration interface that makes it easy for any application or resource manager to integrate with it.

In this chapter we demonstrate that KERMIT can optimize performance of both Hadoop map-reduce jobs and Spark applications using the same integration code and search algorithm. KERMIT optimizes the number of concurrently running containers on-line, in real-time, in response to actual observed container performance, and can achieve a performance of more than 92% of the best possible tuning configuration (exhaustive search of parameter space). Our results demonstrate that on-line automatic tuning can achieve tuning configurations that are up to 30% faster than the basic manual tuning typically applied by big data practitioners. Our technique does not require extensive, time-consuming training runs. It also has a very small code footprint and performance overhead. It does not require any application-specific tuning rules to be developed by application administrators. KERMIT allows application frameworks such as Hadoop map-reduce and Spark to dispense with many manual configuration settings, thereby reducing coding effort for application framework developers and improving end-user experience for administrators.

The remainder of this chapter is organized as follows. Section 5.3 reviews previous work on automatic tuning of big data workloads and discusses key differences between previous work and our approach. Section 5.4.2 first discusses the YARN architecture and how YARN parameters can impact performance of Hadoop map-reduce, Spark and other applications. Section 5.4.2 then describes the KERMIT architecture and how it integrates with YARN. Section 5.5 explains the experimental evaluation methodology, including criteria used to quantify performance improvements and automatic tuning efficiency. Section 5.6 presents our experimental results. Section 5.7 summarizes our key findings and discusses directions for future research.

5.3 Previous Work

There have been many attempts to automatically tune performance of big data workloads. Most of them focused on automatically tuning the Hadoop MapReduce framework. None of the previous papers studied the more general case of auto-tuning

a general purpose resource manager such as YARN. In this section we discuss the relevant related work.

Performance of Hadoop jobs is sensitive to the underlying hardware, the network infrastructure, the JVM configuration and the Hadoop parameter settings. These challenges had been addressed by multiple research groups, revealing the importance of each of the levels [48], [4], [18], [34]. While issues related to the underlying hardware and network infrastructure could be addressed via intelligent infrastructure design, impacts related to the JVM and Hadoop configuration must be attended each time a new map-reduce job is executed. Apache Hadoop exposes over 200 tunable parameters. About 10% of these have significant impact on performance [57], [19], [34]. Manually tuning the performance of Hadoop MapReduce jobs involves trying to experimentally establish the ideal mix of settings for each new job. This is a time-consuming and labor-intensive process. A number of companies specializing on Hadoop deployments have proposed Rule of Thumb (RoT) tuning guides [35], [36], [25]. The difference in job duration (response-time) between the commonly used RoT settings and the optimal configuration can be up to 100 fold [3], [34].

Automatic tuning of Apache Hadoop was explored by multiple research groups. Schaefer and colleagues [48] developed an automatic tuning language for Map Reduce, Atune-IL, that allows a user to explore a selected set of tunable variables, such as the number of threads or the numbers of map and reduce jobs, within the defined user limits. Atune-IL also allows a user to explore the possible performance impact of alternative code block implementations. This approach automates manual tuning to some extent, reduces the search space, and translates the solution search space to a simpler tool. However, it requires a user to have Hadoop tuning expertise in order to select important tunables, and to set up limits and the size of the step for each tunable. The main drawback to Atune-IL is that it requires a developer to learn a new syntax, specific to that language.

HAT, a history-based automatic tuning framework for mapreduce, was introduced by Chen and colleagues in 2013 [6]. HAT tunes the weight of each stage of a map and a reduce task according to values of the tasks in the history. It orchestrates Hadoop back up task execution according to current and historical weights of the tasks. The authors claimed 37% job execution time improvement.

Starfish is a cost-based Hadoop automatic tuning framework developed at Duke

University by Babu and colleagues [19], [18]. The authors succeeded in applying a cost-based approach, popular in relational databases, to optimizing map-reduce performance. Starfish takes into account the different stages of a map-reduce program. It adjusts the tuning at various decision points, which include provisioning, optimization, scheduling, and data layout. The heart of Starfish is the What-If Engine. It employs a combination of simulation and model-based estimation to come up with ideal settings for the Hadoop MapReduce tunables.

In 2013, Liao and colleagues analysed model-based approaches for Apache Hadoop MapReduce optimization and identified a number of major limitations [34]. The research group proposed and implemented Gunther - a search-based automatic tuner for Hadoop MapReduce. The authors evaluated several global search algorithms, and selected a genetic algorithm (GA) for their search implementation. Their modified GA was evaluated on two clusters. Experimental results demonstrated that Gunther achieved near-optimal performance in a small number of trials (<30), and yielded better performance improvements than the rule-of-thumb tuning and the cost-based automatic tuning approaches.

A machine learning approach was explored by Yigitbashi and colleagues [57], who analysed various machine learning-based performance models. Their analysis was conducted on two common Hadoop applications - TeraSort and WordCount - with data sets of various sizes. These authors came to the conclusion that support vector regression exhibits the best performance among machine learning methods. The support vector regression-based automatic tuner was shown to outperform Starfish.

Another search-based evolutionary computation approach was studied by Filho and colleagues in 2014 [37]. They proposed an adaptive tuning mechanism that enabled the setting of specific resources to each job within a query plan. A data structure mapping a job to tuning solution was created based on an analysis of the source code and log files.

All previous work discussed so far are examples of off-line tuning approaches, where changes to the configuration settings are made before or after the job execution. An on-line automatic tuning approach was explored by Li et al. in MRONLINE [33]. MRONLINE automatically tunes a map-reduce job performance as the job runs. This approach has a number of advantages. Each map-reduce job, depending on the type of computation it performs, can respond differently to changes in the tuning parameters. Therefore relying on historical information accumulated from previous jobs as in [6]

can yield suboptimal results. Approaches that rely on modelling [19], [18], or large numbers of training runs [34], are labour-intensive and expensive. MRONLINE focuses on automatically tuning several important parameters of a map-reduce job as it runs. In this respect MRONLINE is the most closely related previous work to our study. MRONLINE focuses on changing MapReduce configuration settings for individual job tasks and correlating these changes with individual task completion times. It works at the Hadoop MapReduce framework level. MRONLINE introduced a specialized search algorithm, called gray-box hill climbing. The algorithm was designed to reduce the number of search iterations needed to find a good-enough solution by allowing the end-user to configure a set of rules that effectively restricted the search space to speed up solution convergence.

The MRONLINE architecture included a monitor, an on-line tuner, and a dynamic configurator. Rather than reading configuration parameters from the `mapred-site.xml` file, map and reduce tasks received their CPU and memory parameters from the dynamic configurator. Different tasks could receive different CPU and memory settings, and could thus display different completion times. The monitor would keep track of task execution times and feed this information to the on-line tuners. The on-line tuner would use a gray-box hill climbing algorithm to find the next set of parameters for the subsequent tasks. MRONLINE performance was evaluated using TeraSort and BBP benchmarks at data scales up to 100GB. The authors claimed 30% performance improvement relative to the default configuration [33].

5.4 KERMIT Automatic Tuning Architecture

Our KERMIT system takes an on-line approach, similar to that used in MRONLINE, but with important differences. MRONLINE introduced dynamic configuration into the Hadoop MapReduce framework. KERMIT introduces dynamic configuration into the more general YARN resource manager framework. Unlike MRONLINE, KERMIT does not keep track of task performance, but focuses instead on the YARN container life-cycle. KERMIT runs as an integral part of the YARN resource manager, intercepts resource requests arriving from the frameworks, and adjusts the actual memory and CPU settings used for containers. This allows KERMIT to be used not only with the Hadoop MapReduce, but also with the other analytic frameworks, such as Apache Spark. In KERMIT we also implement a different search algorithm that does not

require users to configure domain-specific rules. KERMIT is capable of finding close-to-optimal solutions in a small number of iterations. Compared to MRONLINE, KERMIT was evaluated using more complex benchmarks, such as TPCx-HS, and using much larger and more realistic data scales ranging up to 2 TB. KERMIT also introduces a new low-overhead on-line tuning algorithm called the Explorer.

We first review the relevant aspects of the Apache YARN architecture, and then present the details of our KERMIT method.

5.4.1 YARN Architecture Review

This section builds on material presented in Chapter 2, and focuses on aspects that are most relevant to this study. For ease of presentation we discuss the YARN architecture within the context of the Apache Hadoop MapReduce framework v.2.0, which integrates with YARN. Our discussion, however, applies to the other YARN-integrated frameworks as well. The YARN ResourceManager has two main components - the Scheduler and the ApplicationMaster. The Scheduler is responsible for allocating resources to the various applications running on the cluster. The Scheduler uses an abstract Container to encapsulate all tasks and applications. The Container incorporates key system resources needed by the applications - in Hadoop 2.6.0 it supports memory and CPU. The YARN architecture is shown in Figure 5.1.

The ApplicationsManager component of the ResourceManager is responsible for accepting job-submissions, negotiating the first container for executing the application specific ApplicationMaster. The NodeManager is responsible for the containers, monitoring their resource usage (cpu, memory in Hadoop 2.6.0) and reporting it to the ResourceManager/Scheduler. The per-application ApplicationMaster has the responsibility of negotiating appropriate resource containers from the Scheduler, tracking their status, and monitoring their progress.

5.4.2 KERMIT Architecture

Figure 5.2 shows the KERMIT architecture and how it relates to YARN. YARN is designed to support multiple frameworks. Frameworks, such as the Apache Hadoop MapReduce and the Apache Spark connect to the YARN resource manager server. When a framework, for example Hadoop MapReduce, needs to run a job, it sends a resource request to the YARN resource manager. The resource request includes

Figure 5.1: Apache Hadoop YARN architecture. Figure shared under open-source Apache License 2.0 [17].

all of the remaining tasks that need to be run and the desired amount of memory and CPU for each task. For Hadoop MapReduce the desired amount of memory and CPU for each task in the request is determined by the values in `mapred-site.xml` for: (1) `mapreduce.map.memory.mb`; (2) `mapreduce.map.cpu.vcores`; (3) `mapreduce.reduce.memory.mb`; (4) `mapreduce.reduce.cpu.vcores`. The YARN resource manager then uses one of a number of configurable schedulers to determine how much memory and CPU is available on the cluster data nodes to be allocated to this job and determines how many tasks can be executed concurrently on the cluster. YARN creates a Container to track the execution of each task. The Container in the version of YARN used in this study is a virtual construct used to compartmentalize resource allocations for the tasks, and track their life-cycle. The number of Containers created by YARN is determined by the ratio of memory and CPU resources requested by the framework to the total number of memory and CPU resources available for this job on the cluster. Thus the values of map-reduce parameters listed above determine the container density. Container density, in turn, governs the number of JVM processes concurrently executing on the cluster. Optimizing the number of concurrently

Figure 5.2: KERMIT architecture.

executing JVM processes produces performance improvements.

KERMIT is a Java class library that is loaded by, and runs in-process to, the YARN resource manager server. Figure 5.3 shows how KERMIT integrates with the YARN resource manager. KERMIT consists of two main components: (1) AutoTuner; (2) Analyser. KERMIT extends the `ApplicationMasterService` class. It overrides the `allocate()` method, and intercepts the resource request arriving from the framework. The KERMIT AutoTuner then changes the amount of memory and CPU resources requested by the framework for each task and observes the effect using the Analyser. This integration technique has a number of advantages. It allows KERMIT to work with any of the available YARN schedulers and application masters. This is possible because in all cases the resource request will go through the `allocate()` method of the `ApplicationMasterService` class. The request is application-agnostic and contains only the generic desired CPU and memory information. KERMIT focuses on the container response-time analysis and is not aware of any application-specific metrics.

The KERMIT AutoTuner has a sampling interval setting called a Window. The Analyser class maintains a list of Window objects. Each Window object stores the container data collected during the corresponding Window sampling interval. These

Figure 5.3: How KERMIT integrates with YARN.

data include the container start time, the completion time, and the duration. Whenever a container request is made, the Analyser adds this containers start time to the current Window object. Whenever a container completes, which may happen during one of the subsequent Windows, the Analyser finds and updates that containers completion time and duration. The Analyser calculates the average and the standard deviation for these container response-time values. Only the last 20% of the containers in each Window (i.e. the containers that finished last in that Window) are used for these calculations. This approach is designed to reduce the effects of runtime change adjustments lagging behind the dynamic configuration changes.

The AutoTuner uses the Analyser to detect whether a significant change in the container performance has occurred. Significant change is currently defined to occur when the average container duration in the current Window is further apart from the average container duration in the previous Window than the combined container duration standard deviations calculated for those Windows. Once the AutoTuner decides that this significant change has occurred, it uses the Explorer algorithm to perform a search.

The key goals for our Explorer algorithm design were: (1) minimize the tuning overhead by restricting the search space; (2) minimize the administration and configuration. Explorer uses preconfigured ranges for its container memory and CPU values - thus restricting the search space significantly, and reducing the number of search iterations. The administrator needs to only enter the values for container memory

and CPU ranges and specify the duration of the Window. Values for the memory and CPU ranges can be generic values picked based on total memory and CPUs available on data node servers. They can be re-used for different applications.

The Explorer algorithm operates using 3 states: (1) Global search; (2) Local search; (3) Observe. It uses the container statistics from the Analyser to trigger state transitions. State transitions include: (1) Initial to Global search; (2) Global search to Observe; (3) Observe to Local search; (4) Local search to Observe. The algorithm spends most of the time in the Observe state, transitioning to Local search state only when significant change in container performance is detected, as discussed above.

Note that for on-line tuning, it is important to minimize the tuning overhead. Therefore, a simple optimization method with a small number of steps is more desirable. To reduce the tuning overhead, the KERMIT AutoTuner performs the Global search only initially (whenever the resource manager is restarted). Local search is performed only when it detects a change in container performance. The rest of the time it simply observes.

The Explorer starts with the default memory and CPU settings for Hadoop and Spark, and performs a Global search. During the Global search, the Explorer adjusts either memory or CPU value one step up or down in the range. Which dimension to adjust first is chosen randomly. Whether to go up or down is chosen randomly as well. While in the Global search state, the Explorer cycles through all CPU and memory values defined in the ranges using one set of values for one Window. Once all values have been tried, the Explorer picks the best set of settings from its statistics table and switches from the Global search to the Observe state. Each subsequent Window will apply optimal container memory and CPU values determined during the Global search.

State transition from the Observe state to the Local search state is triggered if the average container duration of the current Window is further apart from the average of the previous Window than combined standard deviations of the current and previous Windows. The Explorer randomly picks a dimension and randomly chooses whether to move one step up or down in the range for that dimension. If the change resulted in decreased, or equal, container performance, then the Explorer moves in the opposite direction in the same dimension. If the change resulted in improved performance, the Explorer moves one more step in the same direction. The Explorer will keep moving one step at a time in that direction as long as it sees improvements in container

performance. If the Explorer detects no change after moving in both directions, then it tries to adjust a different dimension using the same logic. If this does not produce an improvement, then the Explorer stops tuning and transitions back to the Observe state using the best CPU and memory values found during the Local search state.

5.5 Evaluation Methodology

Our evaluation methodology focuses on comparing the job end-to-end response-time achieved with KERMIT automatic tuning with two baselines: (1) the basic tuning baseline; (2) the best possible tuning (exhaustive search of parameter space). The reason for using the first baseline was to make sure our approach was yielding a clear, measurable performance improvement relative to simple manual tuning typically performed by big data practitioners. The basic tuning baseline did not use strictly out-of-the-box (OOB) settings. Instead, it was based on a shallow-tuning approach typically used by field practitioners.

To achieve the basic tuning baseline, the OOB Hadoop or Spark configuration was taken as the starting point. Then the `yarn.nodemanager.resource.cpu-vcores` parameter in the `yarn-site.xml` file was set to equal to the total number of CPUs shown by the operating system on each of the cluster nodes. The `yarn.nodemanager.resource.memory-mb` was set to equal to the total amount of memory on each data node. In the `mapred-site.xml` the only parameter that was tuned was the `mapred.child.java.opts` setting which was modified to increase the maximum JVM heap size setting from the default value of 200 MB to 890 MB. This was done to remove the possibility of a memory bottleneck that could gate the performance of both the baseline and the compare, and to make sure that both the baseline and compare had the same JVM heap size, and only the automatically tuned YARN parameters differed. This tuning approach is also similar to the type of tuning commonly performed by field practitioners.

To achieve the best possible tuning baseline, Hadoop MapReduce or Spark settings that were being intercepted by KERMIT and automatically tuned on the YARN side, were searched exhaustively. The settings that were tuned on the MapReduce side are: (1) `mapreduce.map.memory.mb`; (2) `mapreduce.map.cpu.vcores`; (3) `mapreduce.reduce.memory.mb`; (4) `mapreduce.reduce.cpu.vcores`. The Apache Spark parameters tuned for this study were: (1) `spark.executor.memory`; (2) `spark.executor.cores`.

For the CPU settings the values that were tested were between 1 and 8. For the

memory settings the values were tested between 640 and 3200 MB for Hadoop, and between 1024 MB and 12288 MB for Spark. The benchmarks used were Terasort, TPCx-HS and SMB. For TPCx-HS the procedure defined in the TPCx-HS specification was strictly followed. At least 5 data points were collected for each combination of settings to make sure results were repeatable, and the average end-to-end duration or throughput was calculated. Results were plotted to show the shape of the search space. The best combination of settings and end-to-end benchmark duration were used as the best possible tuning baseline for the automatic tuning comparison. For TeraSort and TPCx-HS this procedure was repeated for several data scales 300 GB, 500 GB, 800 GB, 1 TB and 2 TB.

Comparison runs for KERMIT on-line automatic tuning were performed at the same data scale. Several automatic tuning runs were performed to ensure consistency of the results. Comparison focused on three aspects: (1) establishing the improvement compared to the basic tuning baseline; (2) establishing improvement compared to the best possible tuning baseline; (3) establishing how the first two results change with increasing volume of data.

For Apache Hadoop, the TeraSort and the TPCx-HS benchmarks were chosen as the primary workloads for our comparisons. TeraSort is a very commonly used Hadoop benchmarking application. It was chosen due to its widespread use in the Hadoop community. The TPCx-HS benchmark is a recent industry standard. It is inspired by the TeraSort/TeraGen/ TeraValidate utilities that have been commonly used by big data practitioners for several years. TPCx-HS requires that the generate, sort and validate sequence of jobs is executed back-to-back without interruptions. It explicitly disallows manual tuning between the benchmark stages, but it explicitly allows for automatic tuning. Since the HS-Gen, the HS-Sort and the HS-Validate stages perform very different types of processing, it is difficult to find a good combination of tuning parameters that work well for all three of these stages manually. For these reasons TPCx-HS was judged to be a very good benchmark for our study.

For Apache Spark, the recently open-sourced Spark Multi-user Benchmark (SMB) was chosen as the workload. YARN works differently with Spark than with Hadoop. When YARN provides containers to Spark, Spark uses them to start its executors, and schedules all job tasks to run on those executors. Spark executors persist unchanged until the job is finished. Thus for a useful on-line automatic tuning test, we need a multi-job workload. SMB executes concurrently multiple identical Spark TeraSort

jobs, and reports throughput and job duration statistics for these jobs. For this study we used the 2 GB data scale for the Spark TeraSort implementation, and 10 concurrent users for all tests. Our analysis focused on the throughput metric - the most important metric to analytics users. The number of Spark executors was left at default, automatic setting. Spark 1.6.1 was used with YARN 2.6.0.

All measurements were performed on a 4-node cluster comprising 1 management node and 3 data nodes (all bare metal). Each data node was equipped with 1 SSD for the operating system and the Apache Hadoop stack installation, and 4 1.8 TB data disks. Each data node was also equipped with 32 GB RAM and 1 Intel i7 CPU with 4 physical cores running at 1600 MHz. Half of the virtual CPUs visible at the operating system level were disabled via an operating system command to reduce noise caused by the I/O wait. All of the nodes were running the Ubuntu 14.04 operating system.

5.6 Results

Figure 5.4 shows a summary of normalized performance achieved by KERMIT for the Apache Hadoop and Spark workloads. The best possible tuning result for each benchmark was set equal to 1. For all benchmarks, the KERMIT automatic tuning result was observed to be very close to the best possible tuning result. For Hadoop benchmarks TeraSort and TPCx-HS, the best possible tuning result was significantly better than the basic tuning result. For the Spark benchmark SMB, our KERMIT Automatic tuning result achieved an optimal level of performance. By coincidence, the default container size of 1024 MB is optimal for our cluster and data scale, resulting in optimal basic tuning. In general, this is very unlikely, especially for larger clusters. In follow-on studies, where we intend to apply this technique to larger clusters and data scales, we expect that this pattern will change and will be similar to what we observe for other workloads such as TPCx-HS.

Figures 5.5, 5.6 and 5.7 show the shape of the search space for the TeraSort, the TPCx-HS, and the Spark SMB benchmarks respectively, explored via manual tuning experiments conducted to establish the best possible tuning baseline for the container memory. In all cases the data points shown are averages of the data points collected at those memory and CPU settings. The y-axis error bars show the standard deviation of the data. The KERMIT automatic tuning result for the matching data scale is shown as the red marker with error bars that denote the standard deviation of

Figure 5.4: Normalised performance summary for the Apache Hadoop and Spark benchmarks.

the auto-tuning results. The position of the KERMIT auto-tuning data points on the x-axis indicates the container memory value that the Explorer algorithm settled at. For the Hadoop benchmarks, the y-axis records the job duration (response-time). For the Spark SMB benchmark the y-axis records throughput, measured in jobs per hour. In all cases the search space has a global maximum (within the constraints discussed in Section 5.5), as well as the local maxima and minima. In all cases our Explorer algorithm was able to find the global optimum (maximum for the SMB benchmark and minimum for the TeraSort and the TPCx-HS benchmarks), and deliver a result that was statistically very close to this optimum.

Figures 5.8 and 5.9 show a more detailed trace of how the Explorer algorithm operates. The figures show how the Explorer behaved during the SMB workload execution. Figure 5.8 shows the container duration averages collected during all sampling intervals of one of the SMB runs. Figure 5.9 shows how the container memory values were changed by the algorithm during this same SMB run. Although with SMB we are looking to optimize the throughput, the Explorer algorithm seeks to reduce the average container duration observed during its sampling intervals. Reduction in

Figure 5.5: Data for the best possible tuning for TeraSort at the 500 GB data scale.

Figure 5.6: Data for the best possible tuning for TPCx-HS at the 500 GB data scale.

the average of the container durations translates to increased throughput. During the initial explore phase, the Explorer cycles through all of the possible memory values defined in its ranges. These cycles manifest as diagonal alignment of the data points in Figure 5.9. Once the Explorer finds the optimal container memory value, it maintains

Figure 5.7: Data for the best possible tuning for SMB.

this value for all subsequent sampling intervals until it detects a change from the steady state. This manifests as the horizontal alignment of the data points in Figure 5.9. Figure 5.8 shows that the average container durations are systematically reduced after the Explorer finds the optimum container memory value.

The effects of the data scale on the efficiency of the Explorer algorithm were investigated using the TPCx-HS benchmark. Table 5.1 shows the data collected for TPCx-HS at different data scales ranging from 300 GB to 2 TB. Data in the column labelled "Basic Tuning", shows the values collected for the basic tuning baseline discussed in the previous section. Data shown in the column labelled "Best Possible Tuning" shows the values collected for the best possible tuning baseline discussed in the previous section. Data in the column labelled "KERMIT" show the comparison values collected with KERMIT on-line automatic tuning. As expected, the KERMIT data points fall in between the two baselines.

Figure 5.6 shows, in more detail, the data points collected when establishing the best possible tuning for TPCx-HS at 500 GB data scale. For experiments shown in this figure, the values of `mapreduce.map.cpu.vcores` and `mapreduce.reduce.cpu.vcores` were kept at the default value of 1 (experiments where these values were changed

Figure 5.8: The Explorer algorithm optimizing the container durations for the SMB workload.

Figure 5.9: The Explorer algorithm optimizing the container memory allocations for the SMB workload.

from the default value were also conducted). It is interesting to note that when `mapreduce.map.memory.mb` and `mapreduce.reduce.memory.mb` were set to the default

Table 5.1: KERMIT automatic tuning results, compared to the baselines, for the TPCx-HS benchmark at various data scales.

Data Scale (GB)	Basic Tuning (min)	Best Manual Tuning (min)	KERMIT (min)
300	172	159	166
500	291	266	274
800	481	438	449
1000	624	542	566
2000	1350	1165	1179

value of 1024 MB, this resulted in the slowest TPCx-HS performance. The fastest TPCx-HS performance was observed when these container memory configuration parameters were set to 3200 MB. A similar pattern was observed at all of the other data scales that were tested.

Figure 5.10 compares the KERMIT on-line automatic tuning results with the best possible tuning of the TPCx-HS benchmark for data scales ranging from 300 GB to 2 TB. The y-axis shows the improvement (in %) achieved relative to the basic tuning baseline discussed above. The curve for the maximum possible improvement is shown in blue colour. The data points for this curve were collected by performing an exhaustive manual search of the tuning parameter space. The red curve shows the results achieved by KERMIT. Both curves are nearly linear in shape, but the KERMIT curve has a steeper slope, and at higher data scales closely approximates the maximum possible improvement curve.

At small data scales around 300 GB, the improvement achieved even with the best possible tuning is relatively modest. However, it increases with increasing data scale to achieve nearly 14% at 2 TB. Likewise, the improvement achieved with KERMIT on-line automatic tuning is relatively small at small data scales, but grows to nearly 13% at 2 TB. We observe that KERMIT tracks the optimal tuning curve rather closely.

Figure 5.11 illustrates that the performance of KERMIT on-line automatic tuning for TPCx-HS shows a linear increase with increasing data volume. It is important to note that KERMIT was able to demonstrate this at much larger and more realistic data scales compared to previous efforts such as MRONLINE (2TB for KERMIT vs. 100 GB for MRONLINE), and that this improvement is measured against a tuned configuration (basic tuning) rather than the default YARN configuration used

Figure 5.10: Comparison of KERMIT on-line automatic tuning with the best possible tuning for TPCx-HS at different data scales.

for MRONLINE. When extrapolated to larger data scales, the total performance improvement achieved with automatic tuning can be expected to continue to grow. Most actual big data deployment operate at data scales larger than 5 TB. It is unlikely that the performance improvement trend will remain linear as the data scale continues to increase. However, at 5TB and above we expect the performance improvement to be between 20% and 30%.

Figure 5.12 shows the efficiency of KERMIT on-line automatic tuning as a function of the data scale. Here, KERMIT efficiency is defined as the percentage of the best possible tuning performance achieved by KERMIT. For small data scales KERMIT efficiency is about 46% at 300 GB. However, KERMIT efficiency increases significantly with increasing data scale, reaching 92.7% at 2 TB. At larger data scales, KERMIT on-line automatic tuning is expected to be even closer to the best possible tuning.

The main reason why the KERMIT efficiency improves with the increasing data scale is the fact that the total number of containers executed by YARN is directly related to the size of the data that is being processed. With larger data scales each sampling interval used by KERMIT contains more data points that are used to

Figure 5.11: Performance of KERMIT on-line automatic tuning for TPCx-HS appears to show linear increase with increasing data volume.

Figure 5.12: The KERMIT efficiency (percentage of the best possible tuning performance achieved by KERMIT) at different data scales.

calculate the container response time statistics. More importantly with larger data scales the jobs run longer, allowing KERMIT to examine more sampling windows during the job life-cycle. In addition, with the larger number of the sampling windows, the relative cost of Explorer searches decreases as well.

5.7 Conclusion

This chapter presents KERMIT, the first on-line automatic tuning system for YARN. KERMIT optimizes, in real time, the YARN memory and CPU allocations to the individual YARN containers by analysing the container response-time performance. Unlike previous automatic tuning methods for specific systems such as Apache Spark or Hadoop, this is the first study that focuses on the more general case of on-line, real-time tuning of the YARN container density and how this affects the performance of applications running on YARN. KERMIT employs the same tuning code to automatically tune any system that uses YARN, including both Apache Spark and Hadoop. The effectiveness of our technique was evaluated for the Apache Hadoop and Spark jobs using the TeraSort, TPCx-HS and SMB benchmarks. KERMIT on-line automatic tuning can produce configurations that are significantly better than those produced by a human tuning specialist using the shallow-tuning approach typically applied by the field practitioners. In fact, KERMIT on-line automatic tuning was able to achieve performance close to the best possible parameter setting. In our case, we observed 92.7% of the best possible performance for the TPCx-HS benchmark, and close to 100% for the Terasort and the SMB benchmarks. KERMIT configuration is very simple and does not require developers or administrators to develop complex, application-specific rules to produce significant performance improvements. Our Explorer algorithm has a very low overhead and delivers the best results at larger, more realistic multi-Terabyte data scales for Apache Hadoop, and for the larger multi-job workloads for Apache Spark.

Another advantage of on-line automatic tuning over manual tuning is that the on-line automatic tuning can adapt in real-time to changing workload conditions when, for example, as map-reduce analytic jobs progress from the map to the reduce phase, or when a new job in a sequence starts. This results in an overall better and less brittle performance. It is possible to envision KERMIT working alongside application-specific automatic tuners such as MRONLINE [33], with KERMIT focusing on tuning

application parameters common to all applications (such as the memory, the CPU, the network and the disk utilization allotments), and the application-specific tuners focusing on tuning settings that are unique to the respective analytic framework.

The KERMIT on-line automatic tuning running on a resource manager such as YARN dispenses with the configuration settings that are usually left to the user to set, instead delegating the responsibility of finding the optimal settings to the resource manager. This reduces the overall tuning and configuration effort required to deploy analytic applications and the overall cost of ownership.

Chapter 6

Machine Learning Based Spark and Hadoop Workload Classification Using Container Performance Patterns

Performance optimisation is always workload-dependent. A truly autonomic system must be capable of classifying workloads, and adapting its tuning strategy to match the workload type. This chapter addresses the key research question - "Can we accurately classify workloads, and detect changes in important workload characteristics?".

6.1 Problem

For on-line automatic tuning applications - "to tune or not to tune?" - is the key question that the tuning engine needs to be able to answer accurately in order to achieve optimal performance. Too much tuning causes an overhead that can sometime cancel out any performance benefit from optimizing tunable parameters. Not enough tuning results in jobs running slower due to sub-optimal tuning.

A number of researchers studied big data workload characteristics [28], [22], [42]. However, virtually all of these studies focused on lower-level operating system metrics such as CPU, memory and disk utilization, and even lower-level hardware counters such as L1, L2 and L3 cache hit rates. While these data provide important insights about the workload, it is difficult to relate them directly to Hadoop and Spark tunable parameters and develop a tuning strategy at the resource manager level. No published research to date has focused on container performance analysis. However, as explained in Chapter 2, resource managers operate on, and interact with, *containers*. Optimizing

performance of big data applications deployed on a containerized cloud infrastructure requires optimizing the performance of the containers in which they run.

6.2 Contribution

Chapter 2 discussed the architecture of the big data stack, the relationship between the resource manager and the container, and the strategic importance of the container construct. This chapter presents the first experimental study of container performance patterns observed in the Apache Hadoop and Spark workloads. This chapter focuses on the following container performance metrics:

- container duration (response-time),
- container response-time relative standard deviation (RSD),
- container creation rate,
- container completion rate.

The sections below demonstrate that for realistic big data workload sizes (e.g. 2 TB data sets) all important workload changes that are relevant for on-line automatic tuning are accompanied by:

- order of magnitude changes in container creation rate,
- statistically significant changes in RSD.

This chapter presents recommendations for the development of future automatic tuning and machine learning systems. Our experiments demonstrate that the above metrics provide very clear statistical markers that can be used by automatic tuning systems to detect changes in workload characteristics and initialize local and global parameter searches. We also observed that many Hadoop MapReduce and Spark jobs have distinctive signatures that can be used by machine learning systems to identify jobs on the fly and apply effective tuning parameters.

Based on these observations, we built a machine-learning based workload classifier with a workload classification accuracy of 83% and a workload change detection accuracy of 74%. Our observed experimental results are an important step towards developing automatically tuned, fully autonomous cloud infrastructure for big data analytics.

6.3 Previous Work

We begin by summarizing the previous related research into workload characteristics in works focusing on automatic tuning, and then proceed to review the state of the art in big data and cloud workload analysis and characterization.

Wang et al. [50] discuss tuning Spark tunable parameters based on machine learning algorithms, binary classification, and multi-classification. In their model, they tune 13 Spark parameters, such as CPU and memory, on four workloads including Sort, WordCount, Grep, and NaiveBayes. They claim that the Decision Tree algorithm has the best performance for building the model, for making predictions, and for prediction accuracy. The authors use a Recursive Random Search algorithm to find the optimal combination of parameters. The biggest data size in their study is 10 GB. Workloads are single-user. Wang et al. reported that their approach yields an average 36% decrease in execution time. Their study does not contain detailed analysis of workload characteristics.

In 2016, Jia and colleagues [27] focus on automatic tuning of Spark application parallelism for a particular SMT thread count on Power 8 processor. Their machine learning approach has an off-line training stage and an on-line predictive stage. During the off-line training phase, they run different simultaneous multi-threading(SMT) configurations on 20 Spark workloads with eight machine learning algorithms. At the same time, they collect micro-architecture level hardware counter statistics. During the on-line prediction phrase, they tune nine Spark application configuration parameters by invoking prediction hooks. Jia et al. study Spark workload characteristics at the hardware level but do not look at container performance. The authors claim that their method can be generalized to Hadoop and Flink. Although this study touches on Spark executor performance, it does not present an examination of the container performance (Spark executors run in containers).

JellyFish [8] is an on-line tuning method for Hadoop MapReduce. JellyFish introduces the elastic container concept: re-allocating CPU and memory for containers and re-scheduling of the MapReduce tasks. The authors use seven Hadoop workloads, all single-user, with the biggest data size at 100 GB. The authors reported, on average, a 24% performance improvement for first-time job execution, and a 65% performance improvement when a job runs multiple times. Although this method focuses on extending the YARN container, the authors do not include analysis of container performance, and instead focus on optimizing cluster CPU and memory utilization.

MR-Advisor [53] is an off-line tuning tool that can be used for both Hadoop and Spark. It uses a divide-and-conquer algorithm to tune configurations based on the user's input of Hadoop/Spark cluster information, tuning parameters, range of values and benchmark. MR-Advisor proposes a parameter space which maps the input information of the Hadoop MapReduce framework into appropriate configuration parameters. MR-Advisor collects execution time data and uses a divide-and-conquer algorithm to find the optimal result and the corresponding best configuration. In the reported MR-Advisor experiments, the biggest data size is 100GB. This study does not consider, or present, any container-level breakdown of Hadoop MapReduce or Spark job performance.

An algorithm to recommend parameters for Docker containers for Hadoop MapReduce and YARN was studied at IBM by Zhang et al [59]. This research proposed an off-line method for container optimization. The authors chose 6 Docker parameters and 2 YARN container parameters to tune. Based on a KNN algorithm, they select similar past jobs for the current job. Then they rank the performance of past jobs and select a recommended configuration. Zhang et al. study workloads from HiBench, including Bayes, K-means and PageRank. The biggest data size studied was 47GB. The best reported execution time improvement was observed to be 28%. This work focuses on the total job performance and does not provide a detailed breakdown of container-level performance during the course of a job.

Wang et al [51] describe automatic tuning of Hadoop MapReduce from a task slots assignment perspective. Through three experiments on Maximal Clique Finding, TeraSort and WordCount, the authors claim that different workloads have different optimal assignments. This research deals with a pre-container implementation of Hadoop MapReduce and thus is of limited applicability to our study. Slots only describe units of CPU allocation, whereas a container represents a multi-dimensional slice of resources including CPU, memory, disk and network. The authors of this study do not provide a detailed breakdown of slot allocation statistics during the execution of a job.

Jia et al [28] try to characterize the big data workloads and a subset of representative workloads from BigDataBench. They use 45 micro-architectural metrics to evaluate and characterize their experimental results of 32 workloads on either Spark and Hadoop, or Hive and Shark. The data size in these experiments is smaller than 98GB. The authors claim that the software stack has deeper effects on workload behaviours

than the benchmarks and they also find the most related micro-architecture level metrics to support this. This study does not focus on, or describe, container-level performance.

Awan et al [2] compare and characterize batch and stream workloads on Spark. They choose workloads from Spark Core, Spark MLib, Graph X and Spark SQL as representative workloads for batch processing. The authors also focus on micro-architectural level metrics. They claim that the only difference between batch and stream is micro-batching. Batch and stream processing will have the same micro-architectural level behaviour. They also claim that the Spark workloads with DataFrames have better micro-architectural performance than RDD-based workloads. Their study does not consider container-level performance.

HiBench [22] is a realistic benchmark which contains micro-benchmarks, and more realistic applications to evaluate and characterize Hadoop. HiBench was proposed by the Intel China software center in 2010. In [22] the authors use HiBench to characterize the Hadoop MapReduce framework with respect to speed, throughput, HDFS bandwidth, system resource utilization and data access patterns. The biggest data size evaluated in this work is 1TB. The paper focuses on pre-container generation of Hadoop technology, which makes it of limited relevance to our study.

Mulia et al [42] list seventeen categories of cloud workloads. They also connect these categories to system-level metrics such as CPU, network bandwidth, data buses, etc. They mention that based on the low-level metrics such as LLC, IPC, and L1 data cache misses, the change of workload category can be detected. While these researchers provide a fairly deep treatment of the lower-level system metrics, they do not discuss container performance.

Mishra et al [39] propose a method to classify workloads and apply it to the Google compute clusters. The authors claim that coarse-grained and fine-grained are not sufficient to classify the big data workloads based on their experiment and experience. They define three-dimensions (task duration, CPU usage, memory usage) with three qualitative values (small, medium and large) to represent resource utilization of tasks. They use k-means to perform a classification by using the three-dimensions as a feature vector. Their method focuses on tasks and system-level metrics and does not cover container-level performance metrics.

A model of workload resource utilization was proposed by Moreno et al [40]. They use the data from the Google cloud log and characterize workload from both task and

user perspectives. However, they propose a model to characterize the workload resource usage in Google cloud and do not specifically address workload change detection or container performance.

There is an important gap in our current understanding of the big data workload characteristics. The research to date has focused on either total job performance, or on correlating task performance with lower-level system metrics. Containers conceptually fit between jobs and tasks, providing an important, and different, level in the overall performance equation. Containers are more granular than jobs but are coarser than tasks. Modern big data and cloud applications run in containers and, thus, understanding how containers behave during job execution is very important.

6.4 Evaluation Methodology

Our evaluation methodology focused on simulating common Hadoop and Spark workloads and workload transitions using well-understood big data benchmarks. Experiments were planned and performed in increasing order of complexity. Starting with simple single jobs, we then proceeded to the more complex single-user job flows, and then to multi-user job flows. Container performance metrics were compiled by analysing log data.

Table 6.1 summarizes the different workloads and workload transitions analysed in this study, with the benchmarks, data size, and procedure used in each case.

Before capturing container performance statistics for each workload transition experiment, runs were performed to establish the optimal sampling window length. The sampling window duration was chosen so that the majority of windows had a statistically valid number of containers recorded. For example, if all container creation and completion events were recorded during a single, very long, window then this would not make for a compelling analysis.

6.4.1 Container Performance Metrics

As part of our experiments, the following container performance metrics were collected and analyzed:

1. **Container Creation Rate.** This is the number of containers created during a given observation window.

2. **Container Completion Rate.** This is the number of containers that finish execution during a given observation window.
3. **Container Average Response-Time.** This is the average response-time calculated for all containers that complete execution during a given observation window.
4. **Container Response-Time Relative Standard Deviation (RSD).** This metric measures the degree of scatter among container response time measurements in a given observation window. It is defined as the standard deviation of container response-times, divided by the average container response-time for container response-times in a given observation window. Small RSD indicates tightly clustered data while large RSD indicates widely scattered data. Increase in the RSD value across a workload transition can indicate the introduction of a bottleneck due to a change in processing.

Our analysis focuses on calculating both the absolute values for container metrics at steady state, and the relative amount of change that occurs as the workload passes through each transition. The relative amount of change equals the average metric value observed in two observation windows after the transition, divided by the average metric value in the two windows immediately before the transition. In those cases where a drop in the metric value occurred, the change is shown as a decimal fraction. A value of 1 indicates that no change was observed.

6.4.2 Parameter Settings

Unless stated otherwise, the Hadoop MapReduce and Spark configurations used default values. For YARN, the `yarn.nodemanager.resource.cpu-vcores` parameter in the `yarn-site.xml` file was set to the total number of CPUs shown by the operating system on each of the cluster nodes. The `yarn.nodemanager.resource.memory-mb` and `yarn.scheduler.maximum-allocation-mb` parameters were set to the total amount of memory on each data node. In `mapred-site.xml`, the parameter `mapreduce.job.reduces` was set to 36. The parameters `mapreduce.output.fileoutputformat.compress` and `mapreduce.map.output.compress` were set to true. The parameters `mapreduce.output.fileoutputformat.compress.codec` and `mapreduce.map.output.compress.codec` were set to `org.apache.hadoop.io.compress.Default`

Table 6.1: Workloads, workload transitions, and benchmarks.

Transition	Description	Benchmarks, Procedure and Data Size
Hd-sj-1	Transition from map to reduce processing in a single Hadoop map-reduce job.	HiBench WordCount benchmark. 2 TB.
Hd-sj-2	Transition from map to reduce processing in a single Hadoop map-reduce job.	TeraSort benchmark. 2 TB.
Hd-sj-3	Transition from reduce-shuffle to reduce processing in a single Hadoop map-reduce job.	TeraSort. 2 TB.
Hd-suff-4	Transition from TeraGen to TeraSort processing in a Hadoop single-user job flow.	TeraGen-TeraSort-TeraValidate sequence of jobs. 2 TB.
Hd-suff-5	Transition from TeraSort to TeraValidate processing in Hadoop single-user job flow.	TeraGen-TeraSort-TeraValidate sequence of jobs. 2 TB.
Hd-sj-6	Transition from one iteration to another within Hadoop K-Means machine learning job.	HiBench K-Means. 2 TB.
Hd-suff-7	Transition from Hadoop WordCount reduce processing to TeraSort map processing in a single-user job flow.	HiBench WordCount-TeraSort-K-Means job flow. 2 TB.
Hd-suff-8	Multi-user transition from TeraSort map processing in a single-user job flow.	HiBench WordCount-TeraSort-K-Means job flow. 2 TB.
Hd-muff-9	Multi-user transition from TeraSort shuffle to K-Means.	2 users (1 running TeraSort, and 1 K-Means) 2 TB.
Hd-muff-10	Multi-user transition from K-Means iteration back to TeraSort reduce phase.	2 users (1 running TeraSort, and 1 K-Means) 2 TB.
Hd-muff-11	Multi-user transition from TeraSort map phase to K-Means iteration.	2 users (1 running TeraSort, and 1 K-Means) 2 TB.
Hd-muff-12	Multi-user transition from K-Means iteration to TeraSort map phase.	2 users (1 running TeraSort, and 1 K-Means) 2 TB.
Hd-muff-13	Multi-user transition from TeraSort map phase to K-Means iteration.	3 users (1 running TeraSort, and 2 K-Means) 2 TB.
Hd-muff-14	Multi-user transition from K-Means iteration to TeraSort reduce phase.	3 users (1 running TeraSort, and 2 K-Means) 2 TB.
Sp-sj-1	Transition from map() to reduceByKey() processing in a single Spark job.	Spark ARL TeraSort. 2 TB
Sp-suff-2	Transition from Spark K-Means processing to TPC-DS-inspired Q3.	SMB-2 1 user, use case 2 (batch analytics) Spark job sequence. 2 GB per application.
Sp-suff-3	Transition from Spark TPC-DS-inspired Q3 to Q53.	SMB-2 1 user, use case 2 (batch analytics) Spark job sequence. 2 GB per application.
Sp-suff-4	Transition from Spark TPC-DS-inspired Q33 to Q89.	SMB-2 1 user, use case 2 (batch analytics) Spark job sequence. 2 GB per application.
Sp-suff-5	Transition from Spark TPC-DS-inspired Q89 to Q8.	SMB-2 1 user, use case 2 (batch analytics) Spark job sequence. 2 GB per application.
Sp-muff-6	Transition from Spark single-user batch processing to multi-user (3 interactive users).	SMB-2 1 batch user + 3 interactive users, use case 3 (mixed analytics) Spark job sequence. 2 GB per application.
Sp-muff-7	Transition from Spark multi-user (3 interactive users) to single-user batch processing.	SMB-2 1 batch user + 3 interactive users, use case 3 (mixed analytics) Spark job sequence. 2 GB per application.
Sp-suff-8	Initiation of Spark streaming.	spark-perf benchmarking suite.
Sp-suff-9	Completion of Spark streaming.	spark-perf benchmarking suite.
Sp-suff-10	Transition from Spark aggregatedByKey to aggregateByKey(In).	spark-perf benchmarking suite, data scale 3.
Sp-suff-11	Transition from Spark aggregatedByKey to sortByKey().	spark-perf benchmarking suite, data scale 3.
Sp-suff-11	Transition from Spark count() to filter().	spark-perf benchmarking suite, data scale 3.

Figure 6.1: Test-bed topology.

in order to avoid running out of space in the HDFS during bigger runs. The parameter `mapred.child.java.opts` was modified to increase the maximum JVM heap size setting from the default to 850 MB. This was done to remove the possibility of a memory bottleneck impacting container performance. On the Spark side, the `spark.executor.memory` configuration parameter was set to 6G to ensure that most memory on our nodes was utilized.

6.4.3 Hardware and Software

All measurements were performed on a 8-node cluster comprising 1 management node and 7 compute/data nodes (all KVM virtual machines running on IBM S822L Power8 with Dual 10-core Power8 3.42GHz; one bare metal server was used for every two VMs). Each node was equipped with a 100 GB SSD drive for operating system and Hadoop stack installation. All the nodes shared access to a 12TB network shared drive connected through a 10Gb fiber switch. Each node was also equipped with 48 GB RAM and 10 virtual cores. All nodes were running the Ubuntu 16.04 ppc64le operating system. The test cluster topology is shown in Figure 8.5. We used Hadoop 2.7.3 and Spark 2.1.1. In order to facilitate container metric collection, a jar file containing the YARN resource manager and our KERMIT library [13] was built and deployed to replace the standard YARN jar.

6.5 Results

The subsections below discuss how container performance metrics can be used to detect different Hadoop and Spark workloads. We begin by describing steady-state workload characteristics for Spark and Hadoop, as defined by our container performance metrics. We proceed with examining dynamic workload characteristics, i.e. workload transitions produced by each workload. We then present our workload classification and workload transition detection findings.

6.5.1 Steady State Workload Characteristics

Figure 6.2 shows a radar chart that compares the container performance metric averages observed for Hadoop and Spark workloads. To construct this chart, a random sampling of observation windows for Hadoop and Spark observed during steady state conditions were selected for analysis. Container performance statistics, including the maximum, the minimum, the average, and the standard deviation were calculated for all metrics. Although the averages are shown in Figure 6.2, the maximum values were also examined and found to show almost exactly the same trend as the averages.

It was observed that for Hadoop workloads the average container metric values showed much greater range than for Spark workloads. The average container creation rate, container completion rate, container response-time and RSD were all observed to be about 3x greater for Hadoop than for Spark. There is an area on the radar chart where Hadoop and Spark workloads do overlap, but there is a much larger area where they do not overlap.

6.5.2 Dynamic Workload Characteristics - Workload Transitions

Like steady-state performance metrics, changes in the performance metrics produced by big data frameworks when executing jobs and job flows can be used to help identify the type of workload. Figure 6.3 shows a radar chart that compares container performance metric changes observed for Hadoop and Spark workloads. The change of a metric is defined as the average metric value observed after the workload transition divided by the average metric value observed before the workload transition. Figure 6.3 shows the average changes observed for all transitions measured during this study. As for

Figure 6.2: Radar chart showing average Hadoop and Spark workload metric values.

the steady-state performance statistics, full statistics including the maximum, the minimum, the average, and the standard deviation were calculated and examined for all cases. Since the maximum values were found to show almost exactly the same trend as the averages, only the averages are shown.

Hadoop workloads were observed to produce container creation rate changes and container response-time changes that were on average 3x greater than corresponding changes produced by Spark workloads. Changes in the RSD and the container completion rate were observed to show a similar trend. As with the steady-state metrics, an area of overlap between Hadoop and Spark workloads can be observed in Figure 6.3. However, we observe a larger area where workload transition metrics do not overlap.

Figure 6.3: Radar chart showing average Hadoop and Spark workload transition values.

6.5.3 Identifying and Classifying Workloads

To validate whether the container metrics described in this study could indeed be used to identify and classify different workloads, a prototype classifier using several popular machine-learning algorithms was constructed. The prototype was developed in Scala, using Apache Spark Mlib to implement k-means, logistic regression, decision tree, gradient-boosted trees, and random forest algorithms. A machine learning data-set (in libsvm format) was compiled from workload transition data that were labeled as either Spark or Hadoop. The data set was randomly split into training and testing data sets using a 70-30 rule, and the accuracy of prediction for each algorithm was evaluated. The process of splitting, training and testing was repeated 100 times for each algorithm to study the variance produced by the random splits.

Figure 6.4: Workload classification accuracy for common machine-learning algorithms.

The average classification accuracy (and standard deviation) for each algorithm is shown in Figure 6.4. We observe that the Random Forest algorithm achieves the best workload classification accuracy of 83%.

To investigate how different container performance measures affect accuracy of prediction, several additional experiments were performed. The following data sets were prepared: (1) Container creation rate data only. (2) Container creation rate data plus container completion rate data. (3) Container creation rate data plus container completion rate data plus container response-time data. (4) Container creation rate data, plus container completion rate data, plus container response-time data, plus RSD data. The same random split procedure as described above was performed on each data set. The accuracy of classification was evaluated for the Random Forest algorithm. Results are shown in Figure 6.5.

The findings are surprising. We observe that using container creation rate data alone resulted in the best classification accuracy. Adding data from other dimensions reduced rather than enhanced the classification accuracy.

Figure 6.5: Impact of using different container performance statistics on the classification accuracy of the Random Forrest algorithm.

6.5.4 Detecting Workload Transitions

In order to visualize how container metrics change during the course of Hadoop jobs and job flows, data collected for a typical Hadoop single-user job flow are shown in Figure 6.6. This flow executes the following benchmark sequence back-to-back: WordCount-TeraSort-K-Means. Workload transitions are marked with vertical dashed lines and indicated in Figure 6.6. The horizontal axis records the observation window number. The job flow is divided into a series of observation windows. Window number 0 represents the very beginning for the entire job sequence. The duration of each window is fixed (set at the beginning of the job flow). The y-axis records the value of each container metric for a given observation window.

Figure 6.7 shows a multi-user Spark job flow. In this case a single-user thread was started. This thread executed the sequence of batch-type Spark jobs including a K-Means machine learning job and longer-running TPC-DS-inspired queries Q3, Q8, Q53 and Q89. After a delay of 600 seconds, 3 more user threads were started. Each of those user threads executed a sequence of 8 shorter TPC-DS-inspired queries running under a single Spark context. These queries were meant to simulate interactive drill-down operations initiated by a human analyst.

Figure 6.6: Container performance metrics observed for Hadoop WordCount-TeraSort-K-Means job flow, using a 2 min observation window and 2 TB data size.

Figure 6.7: Container performance metrics observed for multi-user Spark workload with batch and interactive query components, using a 30 sec observation window, 2 GB data.

Container metric values measured during the course of a single observation window are shown as different symbols described in the figure legend. As we move right along the x-axis we can see drops and jumps in the patterns of symbols as we cross the workload transitions, represented by vertical dashed lines.

Qualitatively, we can clearly see that the observation window data patterns for the Apache Spark and Hadoop jobs are quite different. The figures discussed above clearly show different frequencies of container creation and completion. The number of containers allocated at one time, as indicated by the y-axis value of the data points, is also different.

Observation window data are time-series data. A time-domain technique was used to evaluate how effectively different container metrics could be used to detect changes in workload characteristics in real-time. Observation window data collected for all data points were replayed as a real-time stream. A rolling average and standard deviation for each container metric were computed for 5 consecutive windows in the stream. During each computation, Welch's test was performed to evaluate whether a statistically meaningful difference existed between the means observed at current and previous steps. Welch's test was performed double-sided, using 95% confidence.

In those cases where a statistically meaningful difference was observed, our prototype code recorded the current observation window and noted a transition there. Transitions identified by the prototype were compared with transitions identified manually by examining YARN, MapReduce and Spark executor logs. Transition detection accuracy was calculated by dividing total transitions identified by the the prototype by total transitions identified manually from logs and multiplying by 100.

Using this technique each container performance metric was separately evaluated for how accurately it could identify important workload changes observed during the execution of Hadoop and Spark job flows. Results are shown in Figure 6.8.

Surprisingly, workload change detection was observed to be the least accurate when using the container creation rate metric (18%), and the most accurate (74%) when using the RSD metric. Changes in nature of processing being performed by containers result in increased variance of the data. This is reflected in different RSD values before and after the transition even in those cases where changes in average container creation rate, container completion rate, and response-time are not statistically significant.

Figure 6.8: Workload transition detection accuracy for different container performance metrics.

6.5.5 Relative Value and Importance of Container Performance Metrics

Based on our findings presented above, it is possible to propose a ranking of container performance metrics:

1. **Container Creation Rate.** This metric was observed to deliver the most accurate workload classification. It is possible to achieve very good classification results using this metric alone.
2. **Container Response-Time Relative Standard Deviation (RSD).** This metric was observed to be less reliable than the other metrics for workload classification. RSD was observed to be very effective, however, for detecting important workload transitions.
3. **Container Average Response-Time.** Although less effective than the first two metrics for both workload classification and workload transition detection, the average container response-time was observed to allow reasonably accurate identification of workload transitions.

4. **Container Completion Rate.** Container completion rate was observed to be less useful than the first three metrics for both workload classification and workload transition detection. This metric was observed to be reliable for identifying completion of jobs, and could be used in conjunction with the other metrics to select the most appropriate tuning strategies. This aspect is discussed in more detail below.

6.6 Implications for the Development of On-Line Automatic Tuning Systems

The container performance metrics discussed above can be used to accurately identify and classify workloads. This is important, because different workloads need to be tuned differently for optimal performance. It is equally important to be able to identify important workload transitions, because changes in workload characteristics often require re-tuning. Techniques described in this chapter can be used to enhance existing and future automatic tuning systems.

One possible architecture for such a system is shown in Figure 6.9. This architecture includes a Transition Monitor component that observes container performance statistics in real time, and detects important workload transitions using the technique described above. The system then uses a Classifier component, which leverages machine learning algorithms to classify the workload. Workload classification is then passed to the Tuner component, which selects an appropriate tuning strategy and searches for optimal configuration parameters.

It is important to point out that this architecture enables tuning approaches that are more fine-grained than tuning job-by-job. Rather than focusing on optimizing end-to-end performance of a single job, on-line automatic tuning systems can use these metrics to optimize overall container throughput in a given observation period. This will enable more granular and adaptable tuning strategies that will ultimately result in higher job throughput in the more realistic multi-user, multi-job conditions.

Figure 6.9: Possible automatic tuning system architecture.

6.7 Conclusion

In this chapter we presented a new way of capturing and analysing workload characteristics of Apache Spark and Hadoop workloads. We demonstrated that it is possible to identify and classify big data analytic workloads with high degree of accuracy using their container performance characteristics. We also demonstrated that it is possible to use container performance metrics to accurately identify important workload transitions.

The most useful metrics were found to be the container creation rate and RSD. Using these metrics, it was possible to accurately distinguish Hadoop and Spark workloads, and identify important workload transitions. Based on these observations, we built a machine-learning based workload classifier and transition monitor with a workload classification accuracy of 83% and a workload change detection accuracy of 74%.

Our observed experimental results are an important step towards developing automatically tuned, fully autonomous cloud infrastructure for big data analytics. The next generation of on-line automatic tuning systems can leverage our findings

to develop tuning approaches that are more fine-grained than tuning end-to-end performance of a job. Resource managers and analytic frameworks can begin to move away from exposing large numbers of tunable parameters, and instead focus on implementing intelligent automatic tuners.

On-line tuners can focus on optimizing overall container throughput at any particular point in time by identifying and classifying workloads, detecting important workload transitions, and selecting tuning strategies that are optimized for a particular workload and situation.

Future research directions will focus on leveraging more sophisticated machine-learning algorithms utilising more dimensions to push workload classification accuracy close to 100%. Container performance metrics can be augmented with CPU, memory and network utilization data to improve accuracy. The goal is to design new low-overhead on-line tuning algorithms capable of achieving large performance gains in complex multi-user multi-job workloads and highly dynamic workload conditions.

Chapter 7

Autonomic Workload Change Classification and Prediction for Big Data Workloads

This chapter addresses the third key research question from Chapter 1 - "Can we predict the future workload characteristics?". To date there has been very little research focusing on detecting and predicting workload changes in big data applications. Sections below recap previous work, outline the theoretical basis for predicting big data workloads, and present the results of our experiments.

7.1 Problem

State of the art approaches from the 'small data' domain rely on past workload history to predict important changes in the workload characteristics. This may be good enough in those situations where the same, or similar, jobs are being executed day to day. In the big data space, however, analytic jobs tend to focus more on exploration and experimentation. This makes big data workloads less cyclical and repetitive than those being run on the more traditional 'small data' systems, making it difficult to leverage the past history to predict the workload type and performance.

Workload change detection and forecasting in big data systems present unique challenges. Analytic frameworks such as Hadoop MapReduce and Apache Spark tend to produce very abrupt changes in workload characteristics as they execute different data processing stages. For example, as a Hadoop MapReduce job transitions from the Map phase to the Reduce phase the nature of the workload changes from

CPU-memory bound to network and disk I/O-bound. When multiple jobs are running concurrently these changes can mask other important workload changes, such as the start or completion of new types of analytic jobs.

Autonomic systems for big data need the ability to forecast at least the near term workload characteristics and job performance without relying on the past history. This information can be used to make more intelligent resource allocation decisions, and adjustments to the tuning parameters of the currently executing analytic jobs and those that are queued and about to start. Longer-term predictions are less useful.

7.2 Limitations of Previous Approaches

To date there has been very little research focusing on the workload change detection and classification in big data workloads. Most research in this area comes from the more traditional Relational Database Management Systems (RDBMS) and Web services areas. These earlier studies rely on analysis of past database query and Web page request patterns to predict workload shifts and cycles, and detect changes by comparing workload characteristics at a given point in time with predicted characteristics. This approach has only limited applicability in the big data space.

Chapter 6 [12] demonstrated a simple technique for workload change detection in Spark and Hadoop workloads. To date this is was the only published study that demonstrated change detection in Spark and Hadoop workloads. However, in order to build intelligent autonomic applications it is not enough to simply detect the change in workload. It is also important for the automatic tuning system to be able to identify and classify the type of change that has occurred, and to predict the type of workload that can be expected in the subsequent observation windows.

7.3 Contribution

This chapter presents a new autonomic pipeline for big data that uses machine-learning algorithms to classify workload transitions as well as workloads, and applies this classification to accurately predict future workload type without relying on past workload history.

We demonstrate 99% change detection accuracy, 90% workload transition classification accuracy, and up to 96% workload type prediction accuracy.

7.4 Previous Work

To date there has been very little research focusing on detecting and predicting workload changes in big data applications. Almost all research studies specifically focusing on workload change detection and prediction come from researchers working on autonomic database management systems (DBMS) in the more established "small data" domain. Most of these works focus on predicting shifts from On-Line Transaction Processing (OLTP) to Decision Support System (DSS) type of workload because these very different workloads require different tunings to achieve acceptable performance. Although big data frameworks have significant architectural differences from the more traditional "small data" DBMS, techniques used for workload classification, change detection and prediction are relevant. Below we discuss the most relevant works in more detail.

Lin Ma et al. [38] describe QueryBot5000 (QB5000) - a system for query-based workload forecasting that can be used to implement autonomic qualities for RDBMS such as MySQL and PostgreSQL. The authors asserted that the best way to model the future workload was to build a model based on the past query type and arrival rate rather than resource utilization. The QB5000 architecture included 3 components: 1 - a Pre-processor that replaced SQL constants with symbols, and used heuristics to reduce millions of unique queries to a more manageable set comprising thousands of query templates; 2 - a Cluster that used an on-line version of DBSCAN non-supervised clustering algorithm to further reduce the total number of models that need to be constructed; 3 - a Forecaster component that used an ensemble comprising a Linear Regression (LR) algorithm and a Recurrent Neural Network (RNN) to predict query arrival rates for cyclical and evolving workloads, and Kernel Regression (KR) to predict query arrival rates for workloads with spikes.

QB5000 used a combination of off-line training and on-line prediction to provide input to the database optimizer. Ma et al. evaluate several prediction horizons. Their results showed improved accuracy when using LR+RNN anseble, or KR for longer prediction horizons or 2 or more days. For short prediction horizon of 1 hr there does not seem to be much difference in accuracy among several techniques evaluated. Their approach relies on past workload history to predict the future query arrival rates, and sudden change in workload pattern would require re-training of the system.

Elnaffar and Martin [10] proposed a framework for predicting shifts in DBMS workload from predominantly OLTP-type to DSS-type. Their framework, called

the Psychic-Skeptic Prediction (PSP) framework, included several components - the Workload Classifier, the Workload Predictor and the Skeptic.

The role of the Workload Classifier was to classify the type of workload at any given point in time based on a feature vector that includes features such as the number of pages read and the number of rows returned for a query. The Workload Classifier uses a technique called Decision Tree Induction to construct its classification model. This technique was chosen over other techniques, such as artificial neural networks (ANN), because it provides for high interoperability and produces a collection of rules that can be readily understood by a human. The Workload Classifier produced a metric called DSSness that expressed the OLTP vs. DSS nature of the workload in a quantitative way.

Elnaffar and Martin noted that running the Workload Classifier continuously resulted in a significant performance overhead [10]. Thus they introduced another component called the Workload Predictor, that would examine a time-series of DSSness, and predict the timing of future workload characteristic shifts from OLTP to DSS. The Skeptic component of PSP would sample the workload characteristics at times close to the predicted shift in the workload characteristics to validate that the prediction was still accurate.

Although the PSP included a mechanism for updating the prediction model in those situations where differences between the predicted and the actual workload characteristics were observed, one main limitation of this approach was that it relied on the past performance to predict the future and thus required significant off-line re-training if a significant change in the workload pattern occurred.

Holze and Ritter [21] presented a continuous, light-weight solution for workload monitoring and workload shift detection using n-gram-models. The authors use the term *workload shift* to describe a number of situations: 1 - introduction of new applications; 2 - sun-setting of obsolete applications; 3 - modifications to existing applications as a result of new releases; 4 - application usage changes caused by, for example, increasing the number of users. Holtze and Ritter [21] contend that *workload shifts* can exhibit short-term and long-term patterns, and focus their study on the detection of long-term patterns.

Huang et al. [23] developed a deep recurrent model for server load and performance prediction in data center. These authors contend that since sequences of user requests

are the root cause of server performance, they should be used as the basis for performance prediction. Their approach uses a type of RNN - the Long Short Term Memory (LSTM) neural network to analyse the server logs containing user requests and predict the future user requests. They use a second LSTM with a Multi-Layer Perceptron (MLP) output layer to predict the performance metrics, such as the throughput and the server CPU utilization based on the workload prediction.

Lei et al. [32] describe a method for detecting hot spots in a virtualized environment. This is an important issue in the cloud space because it affects workload balancing. Their architecture was based on two parts: 1 - an agent that collected system hardware utilization information from the hardware nodes; 2 - a Hadoop MapReduce-based algorithm that identified virtual machines (VM) with anomalous memory utilization. The algorithm scans the time-series of the hardware utilisation metrics collected during the monitoring interval and counts the number of times the average resource utilizations exceed a pre-defined threshold to identify the hot-spots. The importance of hot spot detection is relevant to big data workload analysis because big data analytic jobs run on scaled-out clusters, and the appearance of imbalance in the cluster utilization can denote an important workload change.

Cherkasova et al. [7] proposed a framework for automated anomaly detection and application change analysis. The main objective of their research was to provide a way to pro-actively identify poorly performing enterprise applications. These authors distinguish between the terms *performance anomaly* and *workload change*. They define the term *performance anomaly* to mean that the observed application performance (for example the CPU utilization) cannot be explained by the observed application workload. Their method uses an off-line statistical method - Non-linear Least Squares Regression (Non-negative LSQ) in conjunction with step-wise linear regression to identify significant transactions and model their CPU demand, and an on-line algorithm that computes the new application signature and compares it with the old application signature to identify *performance anomalies*.

Khanna et al. [29] describe a method for autonomic characterisation of workloads using workload fingerprinting. The authors focus on cloud computing and the need for the orchestration layer to forecast changing workload conditions in order to be able to meet Service Level Agreements (SLA). They describe the workload as consisting of a number of phases. Their approach to forecasting consists of two steps: 1 - workload detection and classification; 2 - identifying distinct workload phases. For step 1 they

use the Adaboost [58] ensemble algorithm to synthesize the workload detection model. For step 2 they use the non-supervised K-means clustering algorithm to identify the distinct workload phases. The idea of describing workload as a series of phases or stages is interesting for big data applications because Spark and Hadoop analytic frameworks apply stages of distinct processing.

On reviewing the current state of the art it's possible to make the following generalized observations about the approaches used to identify the workload changes and predict performance:

- there is little standardization in the terminology used to describe workload changes
- researchers use a combination of off-line and on-line techniques
- off-line learning techniques are commonly variants of LR, RNN and LSTM, and Decision Trees and use historical data to construct prediction models
- unsupervised clustering algorithms, such as K-means and DBSCAN, are sometimes used as a pre-processing stage, to reduce the number of prediction models that need to be constructed
- on-line algorithms compare actual executing workload with predictions from historical data to detect workload changes, shifts and anomalies.

The main weakness of this approach is that it relies on the records of past performance to detect change in the currently executing workload and predict the future performance. If the current workload does not match the workloads executed in the past, this approach is not accurate.

In the big data space, analytic workloads often focus on finding new insights and tend to be less repetitive and cyclical than workloads associated with the OLTP Web applications and the more conventional 'small data' RDBMS. Therefore, below we discuss a different approach to workload change detection and performance prediction.

7.5 Workloads and Workload Changes in Big Data Applications

There are many definitions for the term 'workload' in the literature. There are also many different ways to describe workload changes. Due to this, before delving into the theoretic basis and the architecture of our approach, it is important to clarify the terminology and the definitions of these terms as treated in our study.

In this study, the term 'workload' is used to represent any continuous sequence of observation windows with feature vectors that do not show any statistically meaningful differences. Such a sequence represents a period of distinct steady-state processing. Using this definition, for example, a Map stage of a Hadoop MapReduce job would be a distinct workload. Similarly, the reduce-shuffle and reduce stages would be treated as distinct workloads as well. For Spark jobs, the job stages would be treated as distinct workloads from the shuffles.

Genkin et al. [12] define the term *workload transition* to indicate any significant change in workload characteristics. In this chapter we expand on our previous work and provide a more rigorous description, definition, and classification of workloads and workload changes in big data applications.

The term 'workload change' is very broad and can be used to indicate a number of different conditions. We define the following terms to help discuss and analyse changes that can occur in big data systems:

1. *Workload Cycles*. This term refers to regular, repeating workload changes caused by shifts in the usage patterns. This includes, for example, the shift from DSS to OLTP processing, discussed in the section 7.4. These changes can be gradual and symmetrical, or they can be abrupt, asymmetrical, or spiky.
2. *Workload Drift*. This term refers to long-term changes in workload performance characteristics caused by: 1 - changing volume of users; 2 - changing volume of data; 3 - systematic changes in the processing software stack; 4 - changes in the underlying hardware infrastructure.
3. *Workload Anomaly*. This term refers to a sudden, significant, non-repeating, not planned change in the workload characteristics. A sudden spike in the usage activity of a Web commerce site due to an on-line sale is one example.

4. *Workload Transition*. This term refers to a period of non-steady state processing that results when either the usage pattern changes during a *workload cycle*, *workload drift*, *workload anomaly* or as a result of the algorithmic cycles within the analytic framework. One example of this would be the transition from the map phase to the reduce phase during the execution of a map-reduce job.

A Hadoop MapReduce job can present the following sequence of workload transitions during processing:

1. The job start and the beginning of the map phase. This workload transition is typically marked by a sharp spike in disk read activity, high level of container creation and completion, as well as high CPU and memory usage.
2. The completion of the map phase followed by the start of the reduce-shuffle phase. Completion of the map phase is marked by sharp increase in container completion rate, a drop in the CPU utilisation, and an increase in the disk write rate produced by map tasks writing to intermediate files. This is followed by an increase in the container creation rate and a surge in the network read activity as the newly created reduce tasks fetch data from the other nodes.
3. The completion of the reduce-shuffle phase followed by the start of the reduce phase. This event is marked by a marked reduction in the network read rate, and a surge in the disk read rate as the reduce tasks load data fetched from other nodes.
4. The completion of the reduce phase, and of the job itself. This event is marked by a surge in the container completion rate and a corresponding surge in the disk write rate caused by the reduce tasks writing out the final data.

A Hadoop MapReduce, single-user job flow can involve many jobs and many workload transitions described above. Workload transitions for a single-user job flow should show a regular and distinct temporal patterns. For example, the transition marking the start of a job will be typically followed by the transition from the map to the reduce phase.

For multi-user job flows temporal patterns of workload transitions may not be as clear-cut as for the single jobs and the single-user job flows. For example, the start-of-job workload transition may be followed by more start-of-job workload transitions.

Nevertheless, for each start of job transition it is reasonable to expect that a map-to-reduce workload transition will follow.

Although Spark processes data differently than Hadoop MapReduce, similar reasoning can be applied to describe workload transitions that can be produced by that framework:

- The job start. Similar to Hadoop MapReduce job start, the start of a spark job is marked by a surge in container creation activity, and disk read activity.
- The job stage transition. Spark breaks the submitted job down into a number of stages for execution. Unlike Hadoop MapReduce, Spark requests containers at the beginning of the job, and then re-uses them until all of the required processing has been completed. Nevertheless, transitions from one stage of the job to the next can result in significant workload changes. Transition into the shuffle stage is marked by a surge in network read activity. End of the shuffle stage is marked by a surge in disk write, as intermediate results are saved.
- The job end. The result stage of the Spark job is marked by a surge in container completion activity coupled with a surge in disk write activity.

Table 7.2 summarizes the different types of workloads and workload transitions for Apache Hadoop and Spark.

7.6 Architecture

Figure 7.1 shows how the different theoretical constructs in our analysis relate to one another. The automatic tuning engine aggregates real-time streaming data emitted by the resource manager and the cluster infrastructure into observation windows. Each observation window is described by its start time, its end time, and a feature vector. Observation window data are essentially multi-variate, real-time, time-series data. Each feature in the feature vector is a real number.

The real-time stream of observation window data can be treated as a sequence of steady state periods connected to each other by periods of non-steady-state processing, which we term workload transitions. During steady state processing differences in the feature vector data between adjacent windows do not show statistically meaningful

Table 7.1: Workload and workload transition classes and labels.

Transition Label	Description
Workloads	
HDMAP	Hadoop MapReduce Map phase processing.
HDSHUF	Hadoop MapReduce shuffle processing.
HDRED	Hadoop MapReduce Reduce phase processing.
SPSTAGE	Spark job stage processing.
SPSHUF	Spark shuffle processing.
Workload Transitions	
HDJSTART	Start of a Hadoop job. Could be under single or multi-user conditions.
HDMAPSHUFL	Map ends and reduce-shuffle starts.
HDSHUFRED	Reduce-shuffle ends.
HDJEND	Reduce ends, end of Hadoop MapReduce job. Could be under single or multi-user conditions.
SPJSTART	Spark job start.
SPSTSTART	Transition from shuffle to the start of a new job stage.
SPSTSTEND	Transition from a job stage to shuffle.
SPJEND	Spark job end.

Table 7.2: Observation window features and their descriptions.

Feature	Type	Description
numContStarted	Integer	Number of containers created during this observation window.
meanStarted	Double	Average response-time of containers started in this observation window.
rsdStarted	Double	Relative standard deviation of containers started in this observation window.
numContainersFinished	Integer	Number of containers completed during this observation window.
meanFinished	Double	Average response-time of containers started in this observation window.
rsdFinished	Double	Relative standard deviation of containers finished in this observation window.
avgCPU	Double	Average actual CPU utilization of the cluster during this observation window.
averageActiveMem	Double	Average actual active memory utilization on the cluster during this observation window.
avgNetRead	Double	Average actual network read rate on the cluster during this observation window.
avgNetWrite	Double	Average actual network write rate on the cluster during this observation window.

differences. During workload transitions differences in the feature vector data between adjacent observation windows do show statistically meaningful differences. Observation windows that form workload transitions can be described by an additional feature vector that captures the rate of change that has occurred relative to previous observation windows. Each feature in this transition vector is also a real number.

Observation window data typically contain significant amounts of random statistical fluctuation. It is therefore necessary to smooth these fluctuations by applying a sliding

analytical window. As illustrated in Figure 7.1, the sliding analytic window is defined as a set of multiple consecutive observation windows. During our study we used sliding analytic window that aggregated data from 3 observation windows. The analytic window would slide over 1 observation window at each aggregation step. Each analytic window is described by a feature vector containing the average and the standard deviation values for each observation window feature.

Statistically meaningful changes in the workload characteristics can be detected by comparing feature vectors of the current analytic window with the feature vector of the previous analytic window. Feature vectors associated with the steady-state and the workload transition observation windows can be used as the basis for workload classification. This concept serves as the foundation for our architecture.

Our high-level architecture is shown in Figure 8.5, and contains four main components: 1 - *ChangeDetector*; 2 - *WorkloadClassifier*; 3 - *TransitionClassifier*; 4 - *WorkloadPredictor*. We describe the design and the responsibilities of each component below.

The real-time stream of observation window data initially passes through the *ChangeDetector* component. The *ChangeDetector* is a binary classifier that classifies a given observation window as either steady-state or workload transition window. To accomplish this it implements a sliding analytic window that is n observation windows wide. As shown in Figure 7.1 the analytic window covers observation windows $t-2$ to t , where t is the current observation window. With the arrival of each new observation window the analytic window slides over one step.

The *ChangeDetector* is an ensemble of of statistical classifiers. Each statistical classifier performs the Welch's statistical significance test on a single feature from the analytic window's feature vector. It compares the mean and the standard deviation of the feature at observation window t with the mean and the standard deviation at observation window $t-1$. If the Welch's test indicates that there is a statistically meaningful difference, it registers one vote. Each feature classifier also has a weight assigned to it. This weight defaults to 1 and can be optionally adjusted by manual configuration. The *ChangeDetector* counts votes from all 10 feature classifiers and compares the sum of the votes against a threshold parameter. If the number of votes equals or exceeds the threshold, the *ChangeDetector* classifies the current analytic window and the observation window as workload transition windows. The *ChangeDetector* is an unsupervised classifier, and does not require off-line training

Figure 7.1: Theoretical relationship between workloads, workload transitions, observation windows, and analytic windows.

to learn the weights and the threshold. These parameters need to be established experimentally during pre-production testing, and set manually.

If the *ChangeDetector* classifies the current observation window as steady-state, it will pass the observation window data to the *WorkloadClassifier* for further classification into the workload sub-types listed in Table 7.2. If the *ChangeDetector* classifies the current observation window as a workload transition window, it will pass the corresponding analytic window feature matrix data to the *TransitionClassifier* for further classification into one of the workload transition sub-types listed also in Table 7.2. The *TransitionClassifier* uses the analytic window data to calculate the transition feature vector, that is used for classification. This is further described in the paragraph below that discusses the *TransitionClassifier*.

The *WorkloadClassifier* uses the random forest ensemble algorithm to classify the observation windows as workload sub-types. This classifier is trained off-line using a representative training set of the observation windows selected from the previously executed and recorded workloads. It classifies the observation windows in real-time, and outputs one of the workload labels from Table 7.2.

Figure 7.2: High-level architecture for on-line automatic tuning system for big data.

The *TransitionClassifier*, like the *WorkloadClassifier*, also uses the random forest ensemble algorithm, and is also trained off-line. In fact these classifiers share a common abstract base class that wraps the random forest algorithm. Unlike the *WorkloadClassifier*, the *TransitionClassifier* operates not on the observation window, but on the associated analytic window. The sliding analytic window has the same time index as the current observation window, but references several (the number is configurable) previous observation windows in addition to the current observation window. This enables the *TransitionClassifier* to calculate the rate of change for the feature vector. The new derivative vector is used for the real-time classification. The *TransitionClassifier* outputs a transition label from Table 7.2 for each observation window.

The *ChangeDetector*, the *WorkloadClassifier*, and the *TransitionClassifier* are implemented in Scala using Apache Spark 2.4.3 Structured Streaming and MLlib APIs.

After passing through the classification pipeline, the real-time observation window stream containing feature vectors is transformed into a real-time stream of workload and transition labels. This is illustrated in Figure 8.5.

For example, a simple Hadoop MapReduce job that has all relevant stages could

produce a label stream/sequence that reads something like this: "HDJSTART HD-
JSTART HDMAP HDMAP HDMAP HDMAP HDMAP HDMAP HDMAPSHUF
HDMAPSHUF HDSHUF HDSHUF HDSHUF HDSHUFRED HDSHUFRED HDRED
HDRED HDRED HDRED HDRED HDJEND". We now have a stream of data
that very much resembles a textual representation of natural languages, and can be
operated on using algorithmic techniques from that domain.

The *WorkloadPredictor* component uses a Long Short-Term Memory (LSTM) neural network to predict workload and workload transition types that are likely to occur in subsequent observation windows. LSTM neural networks have demonstrated considerable success predicting which words will come next in the real-time natural language processing applications, and the *WorkloadPredictor* uses the same approach. The LSTM is configured with a single layer, and is trained off-line using segments from the recorded workload time-series. During the processing of each current observation window, it predicts a sequence of workload and workload transition classes that can be expected in the next 10 (configurable) observation windows.

The *WorkloadPredictor* is implemented in Python using the popular TensorFlow machine-learning framework. It runs as a separate process and writes predictions into a file that is continuously read by the tuning engine, which uses the predictions to decide whether or not it makes sense to perform parameter searches.

7.7 Evaluation Methodology

Our evaluation methodology focused on simulating common Apache Hadoop and Spark workloads and workload transitions using well-understood big data benchmarks. The experiments were planned and performed in increasing order of complexity. Starting with simple single jobs, we then proceeded to the more complex single-user job flows, and then to multi-user job flows.

Before capturing performance statistics for each experiment, experimental runs were performed to establish the optimal observation window duration. The observation window duration was chosen so that the majority of windows had a statistically valid number of containers recorded. For example, if all container creation and completion events were recorded during a single very long window then this would not make for a compelling analysis.

All metrics were compiled using log analysis. Container performance metrics were

compiled from KERMIT [13] logs. System performance metrics for the cluster were captured using the NMON utility that was executed during the run on all cluster nodes. A Spark application was developed that merged container performance metrics with system metrics, producing an integrated time-series of observation window data. Each observation window feature vector included all container and system performance metrics collected during that window.

A file containing the location of the known workloads and workload transitions was prepared for each time series. The exact time span for a particular workload, and the time at which workload transitions occurred, was established by a Spark application that performed log analysis on YARN logs, Spark event logs, and workload driver logs. The application looked for the key log entries indicating which type of workload was running, or that a workload transition had occurred. The time stamp was then converted to an observation window number, and an entry was added to the known transitions file to indicate the location and the type of workload or workload transition in the time-series. This file was later used to evaluate the accuracy of the classifications and the predictions.

During evaluation each time-series was replayed as a real-time stream of data. The KERMIT streaming engine was implemented as an Apache Spark structured streaming consumer configured to read a file source as a stream. The streaming engine would read the file contents into a streaming data frame. It would then apply SQL and custom stored procedure transformations to the data frame to execute the classification pipeline described in the section 7.6. The result would be a stream of classification labels listed in Table 7.2, and it would be written into a separate text file. This file would be read by the *WorkloadPredictor* component, that would be running as a separate process.

This file containing detected workloads and transitions was then compared with the known workloads and transitions file. The confusion matrix measures, such as accuracy, Positive Predictive Value (PPV) and error rate, were then calculated on the basis of this comparison. To evaluate the workload transition detection accuracy, the transition type was ignored, and detection was treated as a binary classification problem aiming to classify steady state vs. change from steady state. To evaluate workload and workload transition classification accuracy, one vs. the rest approach was used.

When evaluating the supervised learning algorithms, the time series were separated

into a training and testing sub-sets using the 70/30 ratio. Five-fold cross-validation was used to evaluate variance in the classification results and to mitigate over-fitting.

Evaluation measures and approach varied depending on the component. Procedures specific to each component are further discussed below in Section 7.8.

7.7.1 Parameter Settings

Unless stated otherwise, the Hadoop MapReduce and Spark configurations used default values. For YARN, the `yarn.nodemanager.resource.cpu-vcores` parameter in the `yarn-site.xml` file was set to the total number of CPUs shown by the operating system on each of the cluster nodes. The `yarn.nodemanager.resource.memory-mb` and `yarn.scheduler.maximum-allocation-mb` parameters were set to the total amount of memory on each data node. In the `mapred-site.xml` file, the parameter `mapreduce.job.reduces` was set to 36. The parameters `mapreduce.output.fileoutputformat.compress` and `mapreduce.map.output.compress` were set to `true`. The parameters `mapreduce.output.fileoutputformat.compress.codec` and `mapreduce.map.output.compress.code` were set to `org.apache.hadoop.io.compress.Default` in order to avoid running out of space in the HDFS during longer runs. The parameter `mapred.child.java.opts` was modified to increase the maximum JVM heap size setting from the default to 850 MB. This was done to remove the possibility of a memory bottleneck impacting container performance. On the Spark side, the `spark.executor.memory` configuration parameter was set to 6G to ensure that most memory on our nodes was utilized.

7.7.2 Hardware and Software

All measurements were performed on an 8-node cluster comprising 1 management node and 7 compute/data nodes (all KVM virtual machines running on IBM S822L Power8 with Dual 10-core Power8 processors at 3.42GHz; one bare metal server was used for every two VMs). Each node was equipped with a 100 GB SSD drive for operating system and the Hadoop stack installation. All nodes shared access to a 12TB network shared drive connected through a 10Gb fibre switch. Each virtual node was also configured with 48 GB RAM and 10 virtual cores. All nodes were running the Ubuntu 16.04 ppc64le operating system. The test cluster topology is shown in

Figure 7.3: Test-bed topology.

Figure 8.5. We used Hadoop 2.7.3 and Spark 2.4.3. In order to facilitate container metric collection, a jar file containing the YARN resource manager and our KERMIT library [13] was built and deployed to replace the standard YARN jar.

7.8 Results

We discuss our findings in the sections below. The sub-section 7.8.1 discusses binary classification results observed when classifying observation windows as steady-state vs. non-steady state using the *ChangeDetector* component. The sub-section 7.8.2 discusses the results achieved for the *WorkloadClassifier* component. The sub-section 7.8.3 discusses the results achieved for the *TransitionClassifier* component. The sub-section 7.8.4 discusses the results achieved for the *WorkloadPredictor* component.

7.8.1 Change Detection

Evaluation of the *ChangeDetector* component involved the following experiments:

- The experiments to determine the optimal threshold for the *ChangeDetector*, and the weights for feature classifiers.
- The experiments to determine the optimal width of the analytic window.

Figure 7.4 shows the Receiver Operating Characteristic (ROC) plot summarising the results of the experiments performed to establish the best threshold value and

weights. The best results were achieved using a threshold value of 4 votes, and equal weights for the feature classifiers.

After ideal values for the threshold and the weights were established, additional experiments were performed to determine the ideal width of the analytic window. The best results were achieved using the smallest possible analytic window of 3 observation windows.

Figure 7.5 shows the summary of key confusion matrix measures for the *ChangeDetector*. A very high accuracy rate of 0.995 was achieved. It should be pointed out, however, that the observation window data displayed significant amount of class imbalance. There were roughly 100 times more steady-state windows than workload transition windows, and thus the accuracy measure was dominated by the negative case (no statistically significant difference). For this reason we also focused on the Positive Predictive Value (PPV) and the miss rate measures.

The PPV measure was considered to be a more interesting measure for the *ChangeDetector* than accuracy because it evaluates how well the classifier did vs. the positive case (workload transition). As shown in Figure 7.5 PPV was observed to be lower than accuracy. At 0.95 it was still observed to be very good. The miss rate, which evaluates the proportion of actual transitions missed by the classifier, was observed to be low, at 0.05.

7.8.2 Classifying Workloads

The training set for the *WorkloadClassifier* component was assembled manually. Time-series data were separated into training and testing time-series using a 70/30 split, as discussed above. The sample observation window data was extracted from the training time-series, and manually labelled using libsvm format. The observation window training set covered all of the workload classes listed in Table 7.2. The classes in the training set were reasonably well balanced.

As with the *ChangeDetector*, multiple confusion matrix measures were used to evaluate the performance of the *WorkloadClassifier*. Accuracy, PPV and False Positive Rate (FPR) were decided to be the most important measures to focus on. One-vs-the-rest approach was used to calculate the measures for each class. From the class-specific data, average metric value and associated standard deviation were calculated, and are

Figure 7.4: ChangeDetector ROC for different thresholds, and equal weights.

Figure 7.5: ChangeDetector summary of key classifier performance metrics.

shown in Figure 7.6. Experiments were performed to determine the optimal number of trees and categories for the random forest ensemble algorithm used by the classifier. The best results were achieved using 100 trees and 10 categories.

Figure 7.6: Average Accuracy, PPV and FP Rate achieved for the *WorkloadClassifier*.

The *WorkloadClassifier* achieved fairly high average accuracy (0.90) and PPV (0.88), and a fairly low average FPR (0.09). The standard deviation in these measures for all workload classes was observed to be reasonably small (0.02 to 0.06). This is shown in Figure 7.6.

7.8.3 Classifying Workload Transitions

The training procedure for the *TransitionClassifier* component was very similar to that used for the *WorkloadClassifier*. The only difference was due to the fact that the *TransitionClassifier* needs to be trained on the analytic window data, rather than the observation window data.

To accomplish this, segments of observation window data containing known transitions were selected from the training time-series. A utility program was developed to generate and save corresponding analytic window data. The same confusion matrix measures and evaluation approach was used as for the *WorkloadClassifier*.

As with the *WorkloadClassifier*, experiments were performed to establish the ideal number of trees and the categories for the underlying random forest ensemble algorithm. In this case the best results were observed with 200 trees and 20 categories.

Figure 7.7: Average Accuracy, PPV and FP Rate achieved for the TransitionClassifier.

Figure 7.7 shows the summary of the key measures. Results were observed to be good, showing average accuracy of 0.91, and quite similar to those achieved for the *WorkloadClassifier*. This is not very surprising considering that the classifiers use the same underlying algorithm.

7.8.4 Predicting Performance

After evaluation and optimization of the classification pipeline was complete, time-series with class labels given in Table 7.2 were generated for all time series, in both the training and evaluation set. The *WorkloadPredictor* LSTM was trained using complete time-series from the training set.

The testing data were then streamed through the *WorkloadPredictor*, and this component generated a sequence of 10 predicted class labels for each labelled observation window. This sequence was then compared to the actual sequence of labelled windows in the same time series, and accuracy at 3 prediction horizons was calculated. As with the classifiers, one-vs-the-rest approach was used to calculate the accuracy. In this case the workload and the workload transition classes were treated as one set.

Figure 7.8: WorkloadPredictor summary of accuracy by prediction horizon.

The automatic tuning engines and the resource managers that comprise the orchestration layer care mostly about the near-term, and therefore the following prediction horizons were selected for analysis:

- t+1 observation window (i.e. the workload or the transition expected in the next observation window)
- t+5 (i.e. the workload or the transition expected in 5 observation windows from now)
- t+10 (i.e. the workload or the transition expected in 10 observation windows from now)

A summary of the accuracy results observed for the *WorkloadPredictor* are shown in Figure 7.8. As expected, the best accuracy was observed for the closest prediction horizon (0.96). For the longest prediction horizon of t+10 accuracy remained quite high at 0.87.

7.9 Conclusion

This chapter presents the first study focusing on workload change detection, classification, and prediction for big data workloads. Unlike previous studies focusing on the more traditional 'small data' space, our approach does not rely on the longer-term past workload history to predict future workload transitions. Instead we treat big data workloads as sequences of more elemental steady state workload segments connected by non-steady state workload transitions.

We use a classification pipeline to segment multi-variate, real-time workload data and convert it into a sequence of class labels, forming a 'workload language' that is similar in essence to natural languages. We were able to use an LSTM, a very popular natural language processing algorithm, to accurately predict the type of workload, or workload transition, that can be expected to occur in the near to intermediate time horizon. The automatic tuning engines, and the resource managers that comprise the future autonomic big data software infrastructure, can leverage this information to pro-actively manage resource queues, container placement, and reduce the search overhead.

Chapter 8

Zero-Shot Machine Learning Technique for Classification of Multi-User Big Data Workloads

This chapter examines the fourth of the key research questions presented in Chapter 1: "Can we recognize and classify previously unseen, complex, mixed workloads?". In a small departure from the layout of the previous chapters, the sections below begin with a brief introduction. It sets additional context for the advanced machine learning technique that is developed in the subsequent sections.

8.1 Introduction

8.1.1 Background

Previous chapters demonstrated that it is possible to accurately classify different big data workloads using supervised machine-learning algorithms. However, in order to achieve this high level of accuracy, the algorithm has to be trained on a well designed and representative training set. The algorithm hyper-parameters have to be tuned on a validation set. Before being put into production the algorithm's performance has to be evaluated on a test set.

To complicate matters, big data workloads are in fact a broad spectrum of different workload types generated by different numbers of users executing different analytic frameworks on differently structured and sized data sets. When multiple users execute different jobs using the same, or different, analytic framework concurrently - the workload characteristics observed by the resource orchestration layer will differ

significantly from single-job/single-user scenarios.

Nevertheless, it is very important for an autonomic workload optimization system to be able to distinguish multi-user workloads from single user workloads, and to be able to distinguish different multi-user workload combinations among each other. Consider the following scenarios:

- User 1 is executing a Hadoop MapReduce job, which is in the map phase. User 2 is also executing a Hadoop MapReduce job, which is also in the map phase.
- User 1 is executing a Hadoop MapReduce job, which is in the map phase. User 2 is also executing a Hadoop MapReduce job, but it is in the reduce phase.

During scenario 1 both users are executing very similar workloads and the resource demand for CPU would be about the same. Thus, CPU resources should be split evenly between them. During scenario 2, however, User 2 is executing the reduce phase of the job. The reduce phase of an Apache Hadoop MapReduce job typically uses less than 50% of available CPU resources [12]. Thus, up to 50% of additional CPU resource can be allocated to User 1 without impacting User 2 and significantly improving end-to-end job performance for User 1.

The ability to accurately classify multi-user workload scenarios enables autonomic system to select the most appropriate resource allocation algorithms and strategies. However, given that the number of possible multi-user, multi-job workload combinations is very large - how do we train the autonomic system to accomplish this?

8.1.2 Problem

Assembling the training, testing and validation sets is difficult and expensive. This is typically accomplished by manually analysing the workloads running in the production environment, and selecting examples to be used for training the system. This process presents several challenges:

- *Monitoring the production environment workload characteristics.* Even before workload examples can be selected, the production environment needs to be monitored. This requires instrumenting the production environment to enable monitoring of the key workload metrics. This, in turn, implies that the system administrators are aware of which key workload characteristics need to be

monitored. Extensive workload analysis and system administrator training is needed to achieve this.

- *Selecting workload examples to capture.* Once the production environment has been instrumented, extensive manual monitoring and analysis is needed to identify different workload types, and decide which are typical and should be used to train the system.
- *Capturing relevant workload metrics.* Once workload examples have been selected, relevant data has to be extracted from the production environment. This may involve resolving a number of security and privacy issues in addition to the purely technical challenges of extracting data from production databases. System administrators tasked with deploying the autonomic system may not have access to the production data stores. After access to the data has been granted, extracted data typically need to be stripped of any sensitive financial or personally identifiable information before it can be used to train the autonomic system outside the production environment.
- *Transforming workload data into a format that can be consumed by machine learning algorithms.* After the data have been extracted from the production environment they need to be transformed into a format that can be consumed by the machine learning framework. One example transformation could be from a Comma-Separated Value (CSV) format to the frequently used libsvm format.
- *Labelling workload examples.* After the workload data have been transformed into a format that can be consumed by the machine learning framework, they need to be labelled. This involves establishing a labelling scheme for the workloads, and then actually adding labels to the data sets manually, or developing a script to do this.

Clearly, a significant investment in time, software and hardware infrastructure, and personnel training is needed to perform the steps above. Reducing this expense would bring a significant advantage to any organization looking to make their big data and cloud infrastructure more autonomic.

8.1.3 Limitations of Previous Approaches

To date there has been very little research focusing on how to reduce the cost of constructing training sets for autonomic systems. Most research in this area comes from the more traditional Relational Database Management Systems (RDBMS) and Web services areas [10], [34], [21], [23], [38]. These earlier studies rely on analysis of past database query and Web page request patterns to predict workload shifts and cycles, and detect changes by comparing workload characteristics at a given point in time with predicted characteristics. This approach has only limited applicability in the big data space, where workloads tend to focus more on exploration and are less cyclical, and repetitive in nature. In all cases the production workload characteristics are captured and used for algorithm training - subject to effort and expense considerations discussed above.

We are not aware of any prior studies focusing specifically on applying advanced machine learning techniques to reduce training expense and effort for autonomic systems. Although a number of researchers have tried to apply advanced machine-learning techniques, such as Zero-Shot Learning (ZSL), in other domains to address the training expense problem, the general consensus is that the issue is not fully resolved.

ZSL is a subset of heterogeneous transfer learning [52]. ZSL aims to classify instances of unseen classes without being presented with examples of those classes during the training phase. This holds the promise of being able to significantly reduce the expense associated with assembling and labelling training data sets. The practitioner must, however, provide additional semantic information in order to relate unseen classes to seen classes.

Existing ZSL techniques require significant effort to develop the additional semantic information that will augment the training set. This can involve constructing a list of attributes and allowed values, writing the descriptions of classes, or constructing graphs describing the relationships among classes. A more automated approach would help reduce the expense of creating the required additional semantic information needed to apply ZSL.

Another weakness of many current ZSL approaches is that they result in the construction of a classifier that can only classify the unseen classes, but not the seen classes. This introduces additional complexity, because applications using these techniques need to be able to distinguish the seen classes from the unseen classes, and

invoke a different classifier in each case.

8.1.4 Contribution

This is the first study that demonstrates how multi-user workload classes can be described as hybrids of two or more single-user classes, and how a generalized ZSL classifier can be constructed without having to explicitly identify and describe the constituent attributes.

We demonstrate that it is possible to train the autonomic system using only single-user workload examples, and then use it to accurately classify both multi-user and single-user workload scenarios. Our technique can generate a synthetic training data set for the possible multi-user scenarios within seconds to minutes, dramatically reducing the time and expense needed to train the system.

Although our primary motivation is the development of autonomic systems for big data, the hybrid class problem is not limited to this domain. Many classes of objects that we observe and experience in the real world can be thought of, and treated as, hybrids of other, purer, classes. Our technique can be adapted to other workload types and domains.

8.2 Previous Work

This chapter is the first attempt to apply ZSL specifically to workload analysis. Sections below discuss relevant applications of ZSL, and ZSL techniques, developed for other domains.

In their recent paper Wang et al. [52] provide a comprehensive overview and categorization of ZSL research to date. They list 3 works focusing on attribute transfer among classes that are relevant to our technique.

In their study Lampert et al. [31] focus on image classification and describe an approach whereby a previously unseen image class can be classified by learning a number of attributes that comprise the images separately, and subsequently providing attribute descriptions for unseen image classes. The main problem with this approach is that you need to know beforehand which attributes are important, and should be pre-learned. While this may be relatively intuitive with images of animals, in other, more abstract, domains, like workload analysis, it is not.

Al-Halah and Stiefelhausen [1] also focus on image recognition, and propose an approach based on a hierarchical transfer of semantic attributes. Their approach also requires creating per-class attribute descriptions. A hierarchical description of class relationships also needs to be created. Attribute learning happens at different levels of abstraction within the hierarchy. The parent-child relationship among classes is used to propagate attributes to unseen classes. In many situations, however, this does not hold. For example, a Spark workload is not a child of a Hadoop workload, and a common parent (big data workload) is too general to provide meaningful attribute transfer. The class hierarchy can also become very complex and difficult to construct, depending on the domain.

Nam et al. [43] focus on hierarchical relationships among labels in multi-class learning. Their technique involves projecting the label hierarchy into a joint space with instances. The authors claim that label embeddings can be pre-trained. This alleviates one of the limitations in approaches like that used by [1]. Attribute descriptions would still need to be pre-constructed. Some important class relationships would still not be expressed by a label hierarchy.

8.3 The Big Data Workload Spectrum

Chapter 6 demonstrated that workloads produced by different big data analytic frameworks produce distinctive container performance patterns [12]. In our subsequent work [11] we further refined the meaning of the term 'workload'. We defined the term 'workload' to mean "any continuous sequence of observation windows with feature vectors that do not show statistically meaningful differences" [11].

In this work we consider situations when multiple analytic jobs, each with distinct workload characteristics, are running concurrently. The resulting sequence of observation windows - as observed by the resource management and orchestration infrastructure - may appear as a sequence of observation windows that do not show statistically meaningful differences among their feature vectors. The sequence itself though, may appear to be a distinct workload when compared to other sequences. The sequence may be difficult to classify because its feature vectors do not match any examples in the training set.

Figure 8.1 shows how these types of workloads arise. User 1 is executing a Hadoop MapReduce job. The job is currently in its reduce phase of processing. By our

Figure 8.1: How single-user feature vectors combine to form a multi-user feature vector. HDRED and SPSTAGE are defined in [11]. The dashed arrows indicate the aggregation of the observation windows.

definition [11] the reduce phase is a distinct workload characterised by a unique feature vector associated with the corresponding observation windows. The feature vector for this workload is characterised by:

1. Very low container creation and completion rate. Reduce containers are long-running.
2. Long container response-time. For example, the reduce tasks typically take much longer to run than the map tasks.
3. Low CPU utilization. During reduce processing for Hadoop MapReduce jobs the CPU utilization tends to be around 33%.
4. Low network read and write rates. The reduce shuffle is treated as a separate workload using our classification approach.
5. Low disk read rate.
6. High disk write rate. Reduce tasks write the resulting data out to disk.

Concurrently, User 2 is executing an Apache Spark job. This job is also a distinct workload by our earlier definition [11]. The Spark job feature vector is characterized by the following:

1. Low container creation and completion rate. Apache Spark re-uses its executors, which execute in containers, and, therefore, like with the Hadoop reduce workload, these metrics are very low. The container allocation pattern though, will be different than for the Hadoop reduce workload. It could have periodic bursts of container allocations [12] in some of the observation windows.
2. Long container response-times. This metric, again, is similar to what would be observed with the Hadoop reduce workload.
3. Low rate of disk reads and writes. The spark job is executing largely in-memory, writing and reading to and from disk during the shuffles and at the very end of the job.
4. Low rate of network reads and writes. This is similar to point 3 above.
5. High CPU utilization. This depends on what the Spark job is doing, but often the in-memory processing performed by the job will be CPU-intensive.
6. High memory utilization. Spark jobs will typically load data into memory and perform their processing on in-memory data.
7. Low network reads and writes. Shuffles executed between the Spark job stages will typically result in a higher network read and write rate than that observed for the Hadoop reduce workload. However, based on our definition of workload [11] we consider the Apache Spark job stages and shuffles to be separate workload classes, and in this case we are looking at the job stage class.

Because the jobs are running concurrently, the resource/container arbitration software will observe the following features of the combined feature vector:

1. Low container creation and completion rate. This is because both of the concurrent workloads share this feature.
2. Long container response-times. Again, this is because both workloads feature long-running containers.

3. High CPU utilization. This feature will be dominated by the Spark job.
4. High memory utilization. Again, this feature will be dominated by the Spark job.
5. Low network read and write. It will be similar to both the Hadoop reduce and the Spark job stage workloads.
6. High disk write rate. This feature will be dominated by the Hadoop reduce workload.
7. Low disk read rate. This characteristic is common to both workloads in question.

As we can see, the combined feature vector for User 1 and User 2 (see Figure 8.1) will have some features that are very similar in value to each individual workload, although the combined feature vector will be overall distinct from both constituent workloads.

Some features, such as network reads and writes, will be additive - i.e. the combined network read rate will be the sum of both user's workloads. However, in most cases this sum will still be much lower than values that would be observed with a truly network-intensive workload type, such as a Spark shuffle for example.

In other cases the feature of the combined feature vector would be dominated by one of the concurrently running workloads. For example, CPU utilization generated by the Spark job stage workload will often be close to 100%. The Hadoop reduce workload running concurrently, and generating an additional demand for 33% more CPU, will not make the combined CPU utilization higher. Instead the value of this feature will be gated by the bottleneck caused by the physical limitation on available resources. Instead, the response-time of both jobs will increase. Thus, in general the resulting combined feature vector cannot be treated as a simple vector sum of the feature vectors of the two jobs.

In real production environment most workloads will be multi-user workloads. Situations where a single job is running at one time will be fairly rare. The number of users executing jobs at the same time will vary depending on the time of the day, day of the week, the month, or position within the fiscal quarter. Thus, the resulting combined workload feature vectors will display a wide range of values.

We can think of this range of values as a multi-dimensional spectrum of workload characteristics, whereby pure workload types form extreme end points, and combined,

Figure 8.2: Multi-dimensional workload spectrum formed by workload classes defined in [11].

multi-user workloads fill the space in-between. One way of thinking about the workload spectrum is illustrated in Figure 8.2. The pure, single-user, big data workload types described in [11] are shown as end-points at the edges of the radar plot (labelled P1, P2, P3 in the figure). The hybrid, multi-user, workloads with mixed characteristics (labelled H1, H2, H3 in the figure) would fall within the middle area of the plot, and are shown as points connected by lines.

The mixed, multi-user workloads can be thought of as hybrids of the more pure end-point workloads. Because the number of hybrid workloads is essentially unlimited, depending on the number of users and analytic frameworks on the system, sampling the production environment for all possible hybrid workload classes is very difficult. In the sections below we describe the theoretical basis and the implementation details for a new ZSL technique that can be used to effectively classify multi-user workloads without having to sample a wide range of workloads running in the production environment.

8.4 Classifying Hybrid Classes

Figure 8.3 shows how an unseen, hybrid class can be described using two seen, pure classes. In this particular case, to keep the theoretical discussion general, we will use different breeds of dogs as our example. The two pure breeds (seen classes) are: 1 - Golden Retriever; 2 - Poodle. Each dog has a feature vector that describes its characteristics. The feature vectors of the two dog breeds can be sub-divided into two feature sets.

One of the feature sets contains features that show no significant statistical differences between the two breeds. After all, both animals are examples of large dogs, that were bred to assist with hunting. For example, they both have similar size. Both are smart. Both swim well, and both can handle retrieving tasks with proficiency. We call this set of features - the common feature set.

The other set of features is more interesting. This set contains features that differentiate the two breeds. The first obvious feature is colour. Golden Retrievers are, well, golden. Poodles can be of many colours, and commonly can be a mix of more than one colour. Golden Retrievers are also known for their very friendly demeanour towards people and other dogs. Poodles are typically less friendly. One key distinguishing, and highly desirable, feature of Poodles is that they don't shed. They also have a much thinner nose. The values for these features for the two classes will be statistically different. If they were not, then these two classes would not really be different classes. We call this set of features - the distinct feature set.

Because these two breeds of dogs contain distinct, highly desirable features (friendliness, non-shedding), they were bred together to form a new breed called the Goldendoodle - a hybrid of the two pure breeds. The Goldendoodle breed can also be described by a feature vector. Like with the two pure breeds, the Goldendoodle has a common feature set that is not statistically distinguishable from Golden Retrievers or Poodles. Its distinct feature set, though, will show statistical differences from both pure breeds (otherwise it would not be a hybrid). Within its distinct feature set, each feature will be dominated by one of the two pure breeds.

Based on these observations we can propose the algorithm that can be used to implement the semantic space, and the class prototyping function, for both the pure and the hybrid classes. We discuss these in the sub-sections below.

Figure 8.3: How hybrid classes are formed from pure classes.

8.4.1 The Semantic Space

In our case, as in most ZSL applications, the feature space is a real number space where each instance of a pure or hybrid class is represented by a vector: $\chi = \mathbb{R}^D$. We assume that each class instance belongs to only one class. We denote the set of seen, pure classes as $P = \{c_i^P | i = 1, \dots, N_P\}$. Similarly, the set of unseen, hybrid classes can be denoted as $H = \{c_j^H | j = 1, \dots, N_H\}$. Note that $P \cap H = \emptyset$, because otherwise some of the hybrids would actually be pure classes.

Let $D^{tr} = \{(x_i^{tr}, y_i^{tr}) \in \chi \times P\}_{i=1}^{N_{tr}}$ be the set of training instances, where x_i^{tr} is a training instance of a pure, seen class in the feature space, and y_i^{tr} is the corresponding label. Denote the set of testing instances as $X^{te} = \{x_i^{te} \in \chi\}_{i=1}^{N_{te}}$, where each x_i^{te} is a testing instance in the feature space. Denote the corresponding set of labels to be predicted as $Y^{te} = \{y_i^{te} \in P \cup H\}_{i=1}^{N_{te}}$. Note that labels could belong to either pure or hybrid classes because our technique is an example of generalized ZSL.

In order to create our semantic space we need to be able to tell our class prototyping function how the pure classes and the hybrid classes are related. To accomplish this we use a class descriptor, which relates the hybrid class labels to the pure class labels. The class descriptor can be thought of as a set of class descriptions, and can be defined by the following equation: $C = \{c_i = (y_i, \{y_j | y_j \in P\}_{j=2}^{N_{cp}})\}_{i=1}^{N_P + N_H}$.

The equation states that each pure or hybrid class c_i is described by its label y_i , and an associated set of contributing pure class labels $Y_{cp} = \{y_j | y_j \in P\}_{j=2}^{N_{cp}}$. For pure classes, $Y_{cp} = \emptyset$. For hybrid classes, Y_{cp} will have a minimum of two pure class labels. The class descriptor is used in conjunction with the class prototyping function to create the semantic space.

8.4.2 The Class Prototyping Function

The prototyping function for our ZSL classifier is described by the following high-level equation:

$$\pi(\cdot) : P \cup H \rightarrow \mathcal{T}$$

Here $\mathcal{T}$ is the semantic space. Each class prototype within $\mathcal{T}$ is $D \times M$ -dimensional matrix. Here D is the dimension of the feature space χ , and M is the dimension returned by the blending function (discussed below).

The first step in constructing a class prototype during the training phase, is to aggregate the training instances for that class (which are pure, seen class instances). If $D_k^{tr} = \{(x_i^{trk}, y_i^{trk}) \in \chi \times P\}_{i=1}^{N_{trk}}$ is the set of labelled training instances for class k , and the feature vector for $x_i^{trk} = [f_1^i, \dots, f_D^i]$ then the aggregation function $\alpha(\cdot)$ is defined as follows:

$$\alpha(D_k^{tr}) = \begin{bmatrix} (\mu_{f_1}, \sigma_{f_1}) \\ \dots \\ (\mu_{f_D}, \sigma_{f_D}) \end{bmatrix}$$

The aggregation function returns a D -dimensional vector of tuples. Each tuple holds the mean (μ) and standard deviation (σ) for the corresponding feature f . We denote the tuple as ω , and the full vector as $\boldsymbol{\omega}$.

For pure, seen classes, the class prototype matrix will contain this single vector. For the unseen, hybrid classes we need to establish the distinct feature set. The distinct feature set is a set of features that show statistically significant differences in mean values. This is accomplished by applying the Welch's statistical significance test on each pair of tuples among classes. The output is a set of indexes of features that showed statistically significant differences. The reduction function on pure classes k

and l can be defined as follows:

$$r(\omega_k, \omega_l) = \{j_i^{kl} \in \mathbb{Z} | W(\omega_k, \omega_l) = True\}_{i=1}^M$$

The function returns a set of integer feature indexes, for which the Welch's test W is true. The number of features in the set $M \leq D$. All features whose indexes are not in the distinct feature set, are referred to as the common feature set.

Once the set of distinct features has been established, the blending function $\beta(\cdot)$ can be executed. To form the hybrid class prototype, which is a matrix, the blending function averages all ω_i in the common feature set, and selects ω_i from one of the pure classes. If there are M features in the distinct feature set, then there will be $E = M^2 - 2$ possible hybrid prototype vectors (pure cases not allowed). Thus, the output of $\pi(\cdot)$ will be a $D \times E$ sized matrix.

For example, if classes k and l have 4 features, two of which are in the distinct feature set (indexes 1 and 2), and two of which are in the common feature set (indexes 3 and 4), the resulting hybrid class prototype will be:

$$\begin{bmatrix} \omega_1^k & \omega_1^l \\ \omega_2^l & \omega_2^k \\ \overline{\omega_3^{kl}} & \overline{\omega_3^{kl}} \\ \overline{\omega_4^{kl}} & \overline{\omega_4^{kl}} \end{bmatrix}$$

For pure classes, the reduction and blending functions will do nothing, and the class prototype will be a matrix with 1 column and D rows.

The prototyping function for a hybrid class i description that includes pure classes k, l can be written as:

$$\pi(c_i) = \beta(r(\alpha(D_k^{tr}), \alpha(D_l^{tr})))$$

8.4.3 The Synthesizing Function

The synthesizing function uses hybrid class prototypes generated by $\pi(\cdot)$ to create synthetic instances of the hybrid class. In the semantic space, each hybrid class prototype is a $D \times E$ sized matrix, where each column vector represents a possible instance distribution for the hybrid. Each feature ω , in the semantic space, is a tuple

containing the mean and the standard deviation of the mean for that feature in the feature space.

The synthesizing function $s(\cdot)$, generates a random deviation (δ) for each feature mean. The absolute value of δ is less than the standard deviation of that mean (σ). The random deviation can be positive or negative (determined by a coin flip). The synthesizing function adds the random deviation to the mean to create a synthetic instance of the feature in the feature space. For each synthetic feature f^s the synthesizer function can be written as:

$$f_i^s = s(\omega_i) = \mu_i + \delta_i$$

This process is repeated for each feature of each column vector in the class prototype. The resulting output is a set of labelled synthetic hybrid instances in the feature space. More than one synthetic instance can be generated for each column vector in the class prototype, depending on the setting of a configurable hyper parameter.

Once the labelled synthetic instances have been generated for all unseen hybrid classes, any desired machine-learning algorithm can be used to construct the ZSL classifier.

8.5 The ZSL Classifier Architecture

Our ZSL classifier architecture is shown in Figure 8.4 The architecture consists of two components:

- The WorkloadSynthesizer component.
- The WorkloadClassifier component.

The WorkloadSynthesizer component was developed using Apache Spark and Scala. This component includes implementation of the class descriptor, the proto-typing function, and the synthesizing function, as well as the semantic space. The WorkloadSynthesizer runs off-line, as a batch application.

The WorkloadSynthesizer loads a file called the class descriptor (see Figure 8.4) when it is first launched. The pipe-separated class descriptor file is essentially a dictionary listing of all possible pure and hybrid classes that can be encountered by the

Figure 8.4: Architecture of the ZSL workload classifier.

classifier. The class descriptor file represents each pure workload class by its name, or label, which is simply a sequence of letters that can be used to uniquely identify the class. The label for a pure workload class is entered on a single line. The class descriptor file represents each possible hybrid workload by its name, which follows the same rules as for pure classes, and a coma-separated list of pure class labels identifying the classes that can combine to form this hybrid class. The WorkloadSynthesizer then uses the class descriptor to construct an in-memory representation of the semantic space.

In addition to the class descriptor file, the WorkloadSynthesizer needs training data for each pure class that is a part of the semantic space. Once the training data are loaded, the WorkloadSynthesizer invokes another class - the ClassPrototypeBuilder - to construct prototypes for both the pure and the hybrid classes. The ClassPrototypeBuilder implements the class prototyping function, the reduction function, and the blending function described in the section above.

Once the class prototypes have been created, the WorkloadSynthesizer uses the SyntheticInstanceGenerator class to generate a configurable number of synthetic instances for each hybrid class in the semantic space. The synthetic instances are written to a coma-separated value file. Each row of this file contains the hybrid class label (from the class descriptor), and generated values for each feature in the feature

vector. Once this is done, the WorkloadSynthesizer merges the training data set for the pure classes collected earlier with the newly generated synthetic data set for the hybrid classes.

This merged data set is then converted to libsvm format, and used to train the WorkloadClassifier. The WorkloadClassifier is trained off-line, and then used on-line to classify observation windows in real-time. The architecture and performance of the WorkloadClassifier is described in [11]. It should be pointed out that although for this study we chose to use our WorkloadClassifier component developed for earlier studies, once the merged data set containing pure and synthetic hybrid instances is created - any classifier can be used. The classifier is able to classify both pure and hybrid classes. The main innovation developed in this study is our WorkloadSynthesizer component, which implements the core of our ZSL technique.

The overall task flow for using our ZSL classifier is as follows:

1. Execute some simple single-user jobs in the production environment, and record time-series of observation feature vectors for each job.
2. Extract a set of sample observation window data for each pure workload type.
3. Train the WorkloadSynthesizer on pure workload data, and generate synthetic hybrid instances for hybrid, multi-user workload scenarios.
4. Train the WorkloadClassifier on the combined training set containing synthetic instances.
5. Use the WorkloadClassifier on-line, in production to classify observation window data as part of an autonomic workload management system.

Steps 1 through 4 are performed off-line in the intended production or pre-production environment, provided the latter has the same hardware architecture as the former. Step 5 is performed on-line, in real-time.

Benchmarks or sample applications can be used to collect pure workload class data, provided they are reasonably similar to jobs that are executed in production. It should be pointed out again that we are using a very granular definition of workload [11]. Using this granular view it is likely that an observation window collected, for example, during the map phase of a benchmark will be similar to an observation window collected during the map phase of a production job, even though the jobs themselves

Workload Label	Benchmarks
HDMAF	TeraSort (100GB), K-Means, WordCount.
HDSHUF	TeraSort (100GB), K-Means, WordCount.
HDRED	TeraSort (100GB), K-Means, WordCount.
SPSTAGE	ARL TeraSort (100GB), K-Means, SMB-2
SPSHUF	ARL TeraSort (100GB), K-Means, SMB-2.

Table 8.1: Workloads and benchmarks used to collect observation window data for the WorkloadSynthesizer training and validation sets.

could be fairly dissimilar in terms of end-to-end processing that they perform. Thus, classification accuracy on the observation window-level may still be acceptable even though jobs used to produce the training set are different from those running in production.

Below we describe our evaluation methodology, results and future directions.

8.6 Evaluation Methodology

The big data workload data set generated as part of the KERMIT project ([13]) was used to evaluate the hybrid ZSL classifier. The data set consists of multi-variate time-series data collected during the execution of single-user and multi-user Apache Spark and Hadoop benchmarks. Data in each time-series is organised into observation windows, each containing data for 10 features. Workload types and features are described in [11].

Data collected while running the single-user benchmarks were used to assemble the training and the evaluation sets for the WorkloadSynthesizer. Table 8.1 summarizes workload types (originally defined in [11]) and single-user benchmarks used to collect the training set for each. The training set for each workload type included between 100 and 200 observation window feature vectors in libsvm format, which were selected manually from the recorded benchmark time-series. The class of workload for each observation window in the training set was established from the Hadoop and Spark logs. These were used as the training set D^{tr} for the WorkloadSynthesizer.

The WorkloadSynthesizer was used to generate 100 synthetic instances for each ω in the class prototype matrix. These synthetic observation window instances were automatically merged with actual observation window instances collected for pure workload classes earlier. The Random Forest ([20]) based WorkloadClassifier component developed during an earlier study ([12], [11]) was trained using this merged

Figure 8.5: Cluster architecture.

training set. This classifier was used to evaluate the performance of our approach.

Data for the testing set were not taken from the same set as those for the training and validation sets. They were collected by executing a separate set of benchmarks on a big data cluster. The architecture of the big data cluster is shown in Figure 8.5. Although the benchmarks used to generate the testing set were the same, the data scales were different. This was done deliberately to test the hypothesis that the ZSL classifier could be trained using smaller, simpler jobs, and still be able to accurately classify observation window data from more complex production jobs.

Table 8.2 lists both pure (single-user) and hybrid (multi-user) scenarios used to evaluate the performance of our approach. Benchmarks listed in this table for hybrid scenarios were executed concurrently. Pure scenarios were executed sequentially, as a single human user would. Using log information to identify hybrid workload regions of interest, sections of observation windows were extracted from the complete time-series of observation windows collected during the benchmark execution.

The WorkloadClassifier was then used to classify the hybrid, multi-user observation windows. One vs. the rest approach was used to evaluate the accuracy, Positive Predictive Value (PPV), and the False Positive (FP) rate for each hybrid and pure class [56]. A weighted average, based on the proportional number of observation windows belonging to that class, was calculated to obtain the overall result for the ZSL workload classifier.

Training and synthetic instance generation performance of the WorkloadSynthesizer was evaluated using the single workstation with two Intel i7-4510U cores at 2.00 GHz,

Scenario	Procedure
HDMAP-HDRED	User 1: TeraSort (1TB), User 2: TeraSort (1 TB), staggered start.
HDSHUF-HDMAP	User 1: TeraSort (1TB), User 2: TeraSort (1 TB), staggered start.
HDRED-SPSTAGE	User 1: TeraSort (100GB), User 2: ARL Spark TeraSort, staggered start.
HDMAP-SPSHUF	User 1: ARL TeraSort (100GB), User 2: SMB-2.
SPSHUF-SPSTAGE	User 1: ARL TeraSort (100GB), User 2: Spark K-Means.
SPSTAGE	User 1: ARL TeraSort (100GB), SMB-2
SPSHUF	User 1: ARL TeraSort (100GB), SMB-2
HDMAP	User 1: TeraSort (1TB).
HDRED	User 1: TeraSort (1TB).
HDSHUF-HDMAP	User 1: TeraSort (1TB).

Table 8.2: Multi-user (hybrid) and single-user (pure) workload scenarios and benchmarks used to collect observation window data for the WorkloadSynthesizer testing set.

Figure 8.6: The effect of increasing the number of generated synthetic hybrid instances on the accuracy of hybrid workload classification of the ZSL workload classifier.

16 GB RAM, and 1, 1 TB hard disk. The workstation hard disk was formatted with ext4 file system. The workstation was running the Ubuntu 18.04 operating system.

8.7 Results

Figure 8.6 shows the results of the hyper-parameter tuning for the WorkloadSynthesizer component. Optimal results were achieved using 100 synthetic instances per hybrid class. Increasing the number of generated instance further did not improve the ZSL classifier performance.

Figure 8.7 shows a comparison of the average feature vector data for the HDMAp workload phase of two different Hadoop benchmarks - TeraSort and K-Means. Standard deviations for the feature value averages are shown as error bars.

These Hadoop MapReduce jobs perform very different processing. The map phase of the TeraSort job sorts key-value pairs of data. The map phase of the K-Means job implements the portion of the K-Means algorithm that computes the sum of squared distances between data points and centroids. Despite these major differences in processing, the values of the feature vectors collected during the map phase of the two jobs (HDMAp workload class) are remarkably similar. A statistically meaningful difference in values, as indicated by the Welch's test, was observed for only one feature - CPU utilization. Although statistically significant, the difference in average values for this feature is not very large - only about 15%.

Also shown in this figure are the feature vector values (average and standard deviation as error bars, see Figure 8.7) for the HDRED workload generated by the TeraSort job. The feature vectors from this workload show statistically meaningful differences among 5 features relative to both TeraSort-generated, and K-Means-generated HDMAp workload. The proportional differences in values for these feature are also much larger.

When the WorkloadSynthesizer reduction function examined different combinations of single-user (pure) workload classes listed in Table 8.2 the size of the distinct and common feature sets were observed to vary. The average and standard deviation of the size of the distinct and common feature sets were observed to be 4 ± 2 , and 6 ± 2 respectively, out of 10 total features in the feature vector. The key performance metrics - accuracy, PPV and FP rate - were also observed to vary by scenario.

The results for the key ZSL classifier performance metrics are shown in Figure 8.8. Our classifier achieved 90% classification accuracy for the seen, pure workload classes, and 83% classification accuracy for the unseen hybrid classes. The average classification accuracy (pure and hybrid) was 85%. The standard deviation for accuracy was observed to be 5%.

The average PPV for the ZSL classifier was observed as 91% for pure, seen classes, and 83% for the hybrid, unseen classes. The standard deviation for PPV was observed to be 7%.

The average FPR for the ZSL classifier was observed as 11% for the pure, seen

Figure 8.7: Comparison of feature vector values for the HDMAP workload phase generated by the Hadoop TeraSort and K-Means jobs, and the TeraSort HDRED workload.

Figure 8.8: Accuracy, PPV, and FP Rate for the WorkloadClassifier on pure and hybrid workload classes.

Figure 8.9: Scalability of the WorkloadSynthesizer component.

workload classes, and 15% for the unseen, hybrid classes. The standard deviation for the FPR was observed to be 3%.

The ability of the classifier to distinguish hybrid workloads from pure workloads was also evaluated. For this analysis all pure classes were treated as a single combined pure class, and all hybrid classes were treated as a single combined hybrid class. Accuracy was observed to be 92%. PPV was observed to be at 91%. FP Rate was observed to be 9%.

The scalability of the WorkloadSynthesizer component is shown in Figure 8.9. The response-times shown in the figure include training the WorkloadSynthesizer on the training data set assembled for this study, and then generating increasing numbers of synthetic instances per hybrid class. The response-times were observed to scale linearly. Up to 10,000,000 total synthetic multi-user workload observation window instances were generated in under 25 minutes. The training time (time taken to load the observation window feature vectors for pure, single-user workloads and calculate the statistics required by the prototyping and synthesizing functions) was 21 sec.

8.8 Conclusion

This chapter presented the first ZSL technique designed primarily for workload classification. Our technique treats multi-user workloads as hybrids composed of single-user workloads. Our ZSL workload classifier can be trained using relatively simple, single-user workloads, and is able to accurately classify more complex multi-user workload scenarios along with single-user scenarios. Autonomic workload management systems can use this information to optimise their resource allocation algorithms and strategy.

The semantic information needed for our technique - the class dictionary - can be implemented with very little effort. It is simpler than approaches that use class hierarchies, or textual descriptions of class attributes.

Synthetic training data can be generated quickly. Anywhere from dozens to hundreds of synthetic instances are needed to achieve acceptable classification accuracy. These numbers of synthetic instances can be generated in under 5 minutes on a single workstation.

The overall effort and expense involved in training our ZSL workload classifier is much lower than what would be necessary to selectively sample multi-user workload data from the production environment. Our technique also avoids the security and privacy blockers associated with extracting workload data from the production environment.

The accuracy, the PPV and the FP Rate values observed during this study are sufficient for autonomic system development. While it may be possible to achieve greater accuracy by assembling a training set using real multi-user workload data from the production environment, the time and cost savings gained by using our technique would be worth the trade off for many users. Also, training on the production workload data will deliver greater accuracy only if the future workload characteristics will be the same as in the past. If the workload characteristics change significantly, our technique may deliver better accuracy because the WorkloadSynthesizer tries to generate any combination of values in the distinct feature set for a possible hybrid workload, thereby anticipating possible future workloads.

Our ZSL classifier architecture is modular. Any supervised classification algorithm can be trained and used for workload classification after hybrid instances have been synthesized and merged into the training and validation sets. For example, after the WorkloadSynthesizer has run, the combined data set could be used to train a classifier

based on Decision Tree, Bayesian, Random Forest, or an ANN algorithm. This gives our technique greater adaptability to different workload environments, and potential applicability in other domains.

The disadvantage of our technique is that it introduces an additional off-line automated stage into the training process. For many applications this is a reasonable, and acceptable, trade-off.

Chapter 9

Autonomic Architecture for Big Data

This chapter examines the fifth of the key research questions presented in Chapter 1: "Can we propose an architecture that can combine the findings from the first 4 research investigations into a coherent autonomic system for big data analytics?". Sections below delve into the problem in greater detail by decomposing this broader research question into more granular and focused investigations:

- Can we identify common architectural patterns(s) from research performed to date in other domains?
- Can unsupervised machine learning techniques identify distinct workload classes effectively?
- Can we use unsupervised machine learning algorithms to automate creation of labelled training sets for supervised machine learning algorithms to implement an automated model update mechanism?
- Can we use on-line techniques to minimise the amount of off-line training required to implement autonomic architecture features?

9.1 Problem

To date there has been a significant amount of research focusing on autonomic computing. The majority of relevant research has focused on the more traditional, 'small data' RDBMS space [10] [21] [38]. Some research on this area also comes from network, cloud [23] [32], and web [7], but there has been no research to date focusing on big data specifically.

Supervised learning algorithms investigated in Chapter 6 and Chapter 7 require explicit labelling of each workload and workload transition type. The research investigation in Chapter 8 would significantly reduce the required labelling and training effort, but would not eliminate it completely.

We don't want to replace the manual tuning problem with a potentially equally expensive labelling and training problem. We need a solution that can automate labelling and training functions and minimize human intervention.

9.2 Limitations of Previous Approaches

This is the first study that focuses on autonomic performance optimization for big data workloads. Previous works focusing on cloud and traditional small data systems have the following limitations:

1. Coarse view of workload, such as DSS vs. OLTP, makes it impossible to optimize job by job. Big data jobs can have very different optimal configuration parameters.
2. Linear regression models typically used to predict workload characteristics perform poorly with abrupt workload transition common in the big data space.
3. Most methods depend on past workload history to predict future workload characteristics. Coupled with coarse view of workload, this makes them ineffective in situations where workload characteristics change frequently.
4. None of the previous works include the ability to anticipate new, previously unseen, workloads.

Most previous works focus on one of the aspects of autonomic computing but don't describe a complete architecture that implements an autonomic feedback loop.

9.3 Contribution

This work presents the first autonomic architecture specifically designed for autonomic optimization of big data workloads. This architecture implements the feedback loop based on pervasive implementation of machine learning algorithms. The Knowledge

Extraction Resource Management Interface (KERMIT) architecture is able to interact with a wide variety of analytic frameworks and applications regardless of their internal architecture and resource usage pattern.

KERMIT can:

1. Learn new workloads and their characteristics without human intervention.
2. Anticipate the appearance of new, unseen, workload classes.
3. Detect changes in workload characteristics and classify workloads and workload transitions in real-time.
4. Predict which workload types and transitions are likely to occur in the future, and when.
5. Minimize parameter search overhead.

Sections below will discuss previous work on autonomic architectures, and then go into a detailed description of the KERMIT architecture.

9.4 Previous Work

The field of autonomic computing was introduced by IBM in 2001 with the goal of creating computer systems capable of self-management. In 2005 IBM published its reference architecture for autonomic systems [24]. This reference architecture, referred to as MAPE-K, is shown in Figure 9.1. The acronym MAPE-K stands for Monitor(M), Analyze(A), Plan(P), Execute(E), and Knowledge(K).

Since then, there has been considerable research focusing on developing autonomic systems, but this chapter is the first study focusing specifically on autonomic workload optimization for big data. This section summarizes the most relevant studies from the other problem domains. Works focusing on the cloud space and networking are discussed first, followed by more relevant works focusing on small data autonomic databases.

Movahedi et al. [41] present a survey of autonomic network architectures (ANM). The authors classify ANM architectures into hierarchical and flat types. The authors noted that only one of the reviewed architectures - the Cognitive Network Architecture

Figure 9.1: The MAPE-K reference architecture for autonomic systems proposed by IBM. This figure was constructed based on the reference architecture description in [24].

(CNA) - made use of learning techniques. The CNA architecture includes a cognitive plane responsible for data analysis and the decision-making process. They highlighted that the use of learning mechanisms could significantly improve the performance of policy-based adaptation schemes towards finding the optimal solution [41].

Carrera et al. [5] present a study on autonomic placement of batch and transactional workloads. The authors present a technique that enables the existing middleware to fairly manage mixed workloads comprised of both batch analytic jobs and transactional applications. The authors define a simple objective function to measure the difference between the actual response-time and the response-time goal for interactive applications. Their architecture defines the placement control loop and an application placement controller component that periodically inspects the system to determine if placement changes are needed in response to the changing workload. The period of the control loop is configurable. The placement algorithm uses a mathematical model to estimate application performance relative to a given CPU allocation. Their method extrapolates the applications' performance over the duration of the current control cycle and subsequent cycles. The authors claim that their technique improves mixed workload performance while providing service differentiation based on high-level performance goals.

This approach would have limited applicability in the big data space. This is because the mathematical model used to estimate future performance is linear and would not be able to predict the very abrupt workload transitions that big data jobs present, such as the map-to-reduce transition that results in a major change in workload characteristics. Their method does not have any provision for learning. The mathematical model needs to be executed every time, even if similar workload transitions recur.

Gergin et al. [15] describe a decentralized autonomic architecture for performance control in the cloud. Their architecture utilizes feedback loops. It uses a series of autonomic controllers to monitor virtual machine utilization under a Web OLTP-type workload and provisions new virtual machines as needed to achieve SLA objectives. Each controller independently regulates a tier of the application and implements the proportional, integrative, and derivative control laws. The mathematical model underlying each controller uses linear component to extrapolate near-term performance. This approach, as discussed above, would not work well for big data applications because they tend to produce very abrupt workload transitions. Their architecture does not include a learning mechanism.

In their recent paper Nouri et al. [44] focus on a cloud Infrastructure-as-a-Service (IaaS) use case. The authors describe a distributed architecture that aims to maximize performance of a large number of applications deployed on many servers. Each server is a virtual machine. Their view is that a centralized controller would become too complex because it would have to monitor a large number of applications on many server. It may not be able to respond in time when presented with a rapid change in load. A centralized controller would also become a single point of failure in the system.

Their architecture involves deploying an agent on each server. Each agent is responsible for monitoring the application performance on that server. The agents share a common knowledge base. Each agent has application and system monitoring components which feed information to a learning core. The architecture monitors the application response-time statistics and system resource utilization values for the server.

The learning core is based on reinforcement learning. It maps a moving average of CPU utilization values to a set of states for the server. It then uses a utility function that converts that state into a scalar reward value that is used by reinforcement

learning model to select from a set of actions for each state. The states actions and reward values are stored in the knowledge base on each server.

Nouri et al. [44] architecture is only of limited applicability to big data resource management because they consider a scenario whereby multiple applications running on a single server share a pool of resources. Thus exclusive resource allocation to the application is not possible. Containerized big data applications, on the other hand, rely on exclusive allocation of resources.

The architecture described in this work does not have any notion of searching the parameter space to optimize the application performance. Instead the controller can execute a limited set of actions to scale the number of application instances and/or servers up or down. Another aspect that limits this architectures applicability to KERMIT is the fact that this architecture uses linear regression to model system performance. Big data workloads present many abrupt and highly non-linear workload transitions. Nouri et al. architecture is reactive in nature. It does not predict future workload characteristics, and does not anticipate new workload classes.

In recent survey Raza et al. [45] works focusing on autonomic performance tuning in large scale data repositories. This survey explicitly excludes works for focusing on big data. It focuses on the more traditional RDBMS-based data warehouses and on DSS and OLTP workloads. The authors organize research into several categories, including workload classification, performance prediction, and self-adaptation. Most of the surveyed papers focus on one of these aspects and only a few combine them into an architecture that implements the full autonomic cycle. The most relevant ones are discussed in the paragraphs below.

Lin Ma et al. [38] describe QueryBot5000 (QB5000) - a system for query-based workload forecasting that can be used to implement autonomic qualities for RDBMS such as MySQL and PostgreSQL. Although QB5000 was able to forecast future workload characteristics, the authors do not explicitly describe how an autonomic loop could be implemented.

Elnaffar and Martin [10] proposed a framework for predicting shifts in DBMS workload from predominantly OLTP-type to DSS-type. Their framework, called the Psychic-Skeptic Prediction (PSP) framework, included a mechanism for updating the prediction model in those situations where differences between the predicted and the actual workload characteristics were observed - thus implementing the autonomic feedback loop. The main limitation of this approach is that the PSP relied heavily on

past workload cycles to determine whether the model needs to be updated.

9.5 Autonomic Architecture for Big Data Workload Optimization

This section focuses on the different aspects of the KERMIT architecture. Sub-sections below establish a clear description of the key concepts and terms used throughout this work, describe the key autonomic architecture principles, and discuss, in detail, the design techniques used to implement them in KERMIT.

9.5.1 Key Concepts and Terminology

It is very important to establish a clear understanding of the following key concepts and terms before we can delve into the details of the KERMIT autonomic architecture:

1. *Workload*. Different researchers define the term *workload* differently. In this work we use the definition presented during our earlier investigation [11] and summarised in Chapter 7. Consistent with our previous definition, this term refers to a multi-variate time-series of observation windows. Each observation window does not show statistically meaningful differences with neighbouring observation windows. The concept of workload is represented with the symbol Ω .
2. *Workload transition*. A workload transition, like a workload, is a multi-variate time series of observation windows. Unlike a workload, it represents a period of non-steady-state processing. Each observation window of the workload transition will show statistically meaningful changes relative to the neighbouring windows.
3. *Workload drift*. Workload drift refers to changes, whether systematic or random, that can occur to a workload over time. This term is defined in Chapter 7. Sections below will further elaborate on this important concept.

The broader set of processing performed by the system can be described as a sequence of workloads connected together by workload transitions. This is shown in Figure 7.1 in Chapter 7.

This view of workload is more granular than that generally used by other researchers (for example in [21], [10]). Nevertheless, it allows for a more systematic and automated treatment of workload analysis and optimization.

9.5.2 Autonomic Architecture Principles

The MAPE-K architecture, shown in Figure 9.1 implements a feedback loop designed to achieve the key properties essential for an autonomic systems, as defined by IBM. These properties were defined as:

- *Self-configuration.* An autonomic system must be able to configure, or re-configure, itself without requiring human intervention.
- *Self-healing.* In the event of a fault, an autonomic system must be able to find a way to mitigate the fault without human intervention.
- *Self-optimization.* An autonomic system must be able to optimize its performance within specified goals without human intervention.
- *Self-protection.* An autonomic system must be able to defend itself against misuse without human intervention.

Within the framework of the MAPE-K reference architecture (see Figure 9.1) the Autonomic Manager uses Sensors to Monitor a Managed Resource. Monitoring generates Knowledge that pertains to the Managed Resource. This knowledge is Analysed and used to Plan the system's response to changes. During the Execute phase the Autonomic Manager uses the Effectors to affect the Managed Resource in accordance with the Plan.

The research presented in this chapter focuses on achieving the self-optimization characteristic. By extension some aspects of the self-configuration, self-healing, and self-protection characteristics are drawn in as well. This is due to the fact that in order to change the performance characteristics of the big data stack changes to the configuration are required.

The self-healing and self-protection properties were not directly the subject of this investigation. Nevertheless, it can be said that the KERMIT architecture partially addresses those properties as well. For example, failure of one or more nodes in the cluster can present itself as the appearance of new workload types because the

Figure 9.2: Mapping the KERMIT sub-systems and components onto the MAPE-K reference architecture.

loss of the CPU and memory resources will alter the observed feature vector of the observation windows. The KERMIT architecture, as described below, will be able to react to this and find a new optimum. This type of response can be characterised as partial self-healing. The self-healing is partial in this case because KERMIT does not address bringing replacement nodes on-line. Similarly, KERMIT's ability to respond to new workload types can be viewed as partial protection from a sophisticated type of DoS/DDoS attack targeting back end systems.

Sections below describe how the KERMIT architecture layers map onto the MAPE-K reference architecture, and describe key components and algorithms in detail.

9.5.3 The KERMIT Architecture

Figure 9.2 shows the mapping of the key KERMIT sub-systems and components onto the MAPE-K reference architecture. The KERMIT architecture is broadly divided into two subsystems: 1 - On-line; 2 - Off-Line.

The On-line sub-system operates in real-time, and includes the Workload Monitoring sub-system, the Agents sub-system, the Dynamic Configurator sub-system, and the Workload Classification, Prediction and Optimization sub-system. The Off-line

Figure 9.3: The high-level KERMIT components shown in relation to the resource manager and the big data cluster.

sub-system operates asynchronously in batch mode, and includes the Workload Discovery and the Workload Characterization sub-systems. All KERMIT sub-systems both update, and read from the Workload Knowledge Base sub-system.

Figure 9.3 shows the high-level components of the KERMIT architecture, and how it relates to a typical big data cluster. The On-line sub-system interfaces with the Resource Manager process (RM Process in Figure 9.3) using a plug-in (KPlg). The core of the Workload Monitoring sub-system is the KERMIT Workload Monitor (KWmon) component. This component is a streaming engine that receives messages from the KPlg component and from the KERMIT system monitoring agents (KAgnt) deployed on cluster nodes.

The KERMIT Workload Analyser (KWanl in Figure 9.3) is the main component in the Off-line sub-system. The KWmon and the KWanl components implement real-time and batch machine learning pipelines form the core of the KERMIT architecture. The details of each sub-system architecture are described in the sections below.

9.6 The KERMIT On-line Sub-System Architecture

The KERMIT autonomic architecture stresses real-time, on-line workload optimization. It strives towards these key goals:

- **Maximize responsiveness.** If the system is able to quickly react to changing workload conditions, then the need for far-horizon workload prediction and resource planning (as in [10]) is reduced.
- **Reduce search overhead.** Excessive searching for the optimal configuration can not only negate the positive effect of having a better configuration, but actually slow down the performance of the system.
- **Eliminate manual configuration.** Components that comprise the KERMIT architecture try to keep the number of hyper-parameters to an absolute minimum, and their values are chosen so as to avoid frequent adjustments.

The KERMIT on-line sub-system implements a real-time machine learning pipeline capable of performing on-line change detection, classification and prediction. The theoretical bases and the operation of this pipeline is described in Chapter 7 and [11]. This chapter focuses on explaining the interaction between the KERMIT on-line sub-system, the Workload Knowledge Base (KWknb) and the KERMIT off-line subsystem.

This sub-system also includes the KERMIT plug-in component. This component uses information generated by the KERMIT workload monitor to optimize the resource allocation to big data frameworks such as Apache Spark and Hadoop MapReduce.

The sub-sections below first focus on explaining the workload monitoring interactions. Then the interactions involved in classification, prediction and optimization are explained.

9.6.1 Workload Monitoring

The KERMIT workload monitor component (KWmon in Figure 9.3) is a streaming engine that aggregates, in real-time, system information from the cluster servers and container performance information from the cluster resource manager.

The KERMIT agents (KAgnt) are started on all cluster nodes during the resource manager start-up. They collect system utilisation data at a specified interval. This

interval relates to the KERMIT hyper-parameter that controls the observation window duration. It is typically set to one tenth of the observation window duration. The KERMIT agents send messages that are consumed by the KERMIT workload monitor streaming engine. The current KERMIT implementation uses text files stored on a shared distributed file system to share data with KWmon, but a messaging platform such as Apache Kafka can be used as well for a more distributed implementation.

The KERMIT plug-in functions as a monitoring agent as well. It collects container performance information whenever invoked by the resource manager, and sends messages to the KWmon component. Currently, as with the KAgnt components, this is done by populating a text file on a shared file system, but a messaging system can be used for a more distributed implementation.

The KWmon component aggregates data from the KERMIT plug-in and the KERMIT agents, and stores the aggregated data in the KERMIT workload knowledge base (KWnb in Figure 9.3). The aggregated data is the observation data, captured as a multi-variate time series described in Chapter 7 and in [12] and [11].

The KERMIT workload monitor component also interacts with the KERMIT workload knowledge base and the KERMIT plug-in to implement the machine learning pipeline critical for workload classification and prediction, and the configuration parameter space search. These interactions are discussed in detail in the next subsection below.

9.6.2 Workload Classification, Prediction, and Optimization

Figure 9.4 shows the logical architecture of the KERMIT workload knowledge base. The KERMIT workload knowledge base is implemented using a shared distributed file system, such as HDFS. The KERMIT workload knowledge base contains a Landing Zone (LZ in Figure 9.4), a Transformation Zone (TZ), and an Analytics Zone (AZ).

The raw time-stamped data generated by the KERMIT agents and the KERMIT plug-in components are stored in the LZ. These data are mostly loosely structured text files or log files. There is one file for each agent, and one for the KERMIT plug-in. The KERMIT workload monitor reads these data in real-time, treating each file as a streaming source, as new time-stamped data are appended. It transforms the time-stamped data into a structured format and aggregates them into observation windows O_t with the associated feature vector F_t .

Figure 9.4: Logical architecture of the KERMIT knowledge base.

The KERMIT workload monitor applies the workload classification pipeline described in Chapter 7 and [11] to transform the input stream of observation windows $\{O_t\}_{t=1}^n$ into a stream of labels $\{Y_t\}_{t=1}^n$, and writes out a sequence of workload context objects $\{C_t\}_{t=1}^n$.

The workload context at observation window t , C_t , contains the following information:

- The workload label for the current observation window t .
- The predicted workload label for time horizon $t+1$.
- The predicted workload label for time horizon $t+5$.
- The predicted workload label for time horizon $t+10$.

The KERMIT plug-in code is called whenever the resource manager responds to a resource request from an analytic framework. The integration between the KERMIT plug-in and the resource manager is described in Chapter 5 and in [13]. The KERMIT code intercepts the resource managers' response to the analytic frameworks' resource request and engages the low-overhead, conceptually simple, Explorer search algorithm to find an optimal configuration [13]. The explorer algorithm is described in detail in Chapter 5.

The Explorer demonstrated that it was capable of achieving up to 30% better performance than rule-of-thumb tuning by a human practitioner, and up to 92.5% tuning efficiency relative to the best possible tuning by a human practitioner. However, many real-world workloads are repetitive in nature. For example, the job to tally up the daily financial results is run at the same time every day. Some jobs are executed many times daily. If we consider that our definition of the term workload is more granular than a job - it stands to reason that the same workload type may be encountered many times during the day, or even the hour.

It only makes sense to enhance the KERMIT plug-in architecture, described as the KERMIT Architecture in Chapter 5 and in [13], with the ability to recognize workloads that have already been executed, and for which the optimal configuration has already been found. This allows KERMIT to avoid repeating the same parameter search multiple times, and achieve further performance gains under realistic workload conditions.

The KERMIT plug-in extends the KERMIT Analyser component presented in Chapter 5 and in [13] with the capability to read the workload context stream generated by the KERMIT workload monitor. When called by resource manager it first checks the workload monitor stream output and retrieves the workload context object. It then checks the label of the currently executing workload, and retrieves the workload descriptor object from KERMIT workload knowledge base.

The workload descriptor object contains the following items of information:

- Statistics for each feature in the workload feature vector (described further below).
- Centroid values for the workload (described further below).
- A true or false flag indicating whether or not an optimal configuration has been found.
- A set of configuration values to be used.

The main high-level algorithm used by the KERMIT plug-in is described in Algorithm 1. When the resource manager calls the KERMIT plug-in code in response to a resource request from one of the analytic frameworks, the Analyser component reads the workload context stream $\{\mathbb{C}_t\}_{t=1}^n$. It reads in the latest context $\mathbb{C}_t$, and compares the current time with the observation window associated with the context

to make sure that the plug-in and the KERMIT workload monitor are in-sync. If they are not then an error is logged and a default configuration is used until the error is resolved.

Once the context has been read in, the plug-in checks the workload type label for the current observation window $\mathbb{C}_t.currentWindow$. When the KERMIT workload monitor first starts and the different workload types have not yet been determined, the type will be UNKNOWN. In this case the KERMIT plug-in will simply use the default configuration $\mathfrak{J}^D$ as the optimal configuration $\mathfrak{J}_i^o$ for this workload. The plug-in will wait until the off-line sub-system workload discovery catches up, and will continue to check the current workload type each time it is called by the resource manager.

Once the KERMIT plug-in encounters a current workload label that is known, it will check the KERMIT WorkloadDB to see if an optimal configuration for this label has already been established. If so, it will simply retrieve this configuration from the WorkloadDB.

If there is no optimal configuration associated with this workload, then the plug-in will check if workload drift has been detected. If so, then there will be a configuration in the WorkloadDB associated with this workload label, but this configuration will not be optimal. In this case the KERMIT plug-in will retrieve this configuration and pass it to the Explorer algorithm to initiate a local search described in Chapter 5 and [13]. Once the local search finds the optimal configuration, the plug-in will update the KERMIT WorkloadDB with this configuration, and set the optimal configuration field in the database to the value "True".

If workload drift has not been detected then the WorkloadDB will not have a sub-optimal configuration stored for the workload because the workload has just been detected by the off-line sub-system. In this case the KERMIT plug-in will start the Explorer algorithm's global search described in 5 and in [13]. Once the global search finds the optimal configuration the KERMIT plug-in will update the WorkloadDB with this configuration, and set the optimal configuration field in the database to the value "True".

The algorithms used to identify different workload types and detect workload drift are discussed in the section below.

Algorithm 1 Main KERMIT plug-in algorithm.

Input: $\{\mathbb{C}_t\}_{t=1}^n \neq \{\}$ {Workload context stream must be started and available.}

Input: $O_t \neq \emptyset$ {Current observation window.}

Output: $\mathfrak{J}_i^o$ {The optimal configuration for this workload}.

 if $\mathbb{C}_t.currentLabel$ is UNKNOWN **then**

 $\mathfrak{J}_i^o \leftarrow \mathfrak{J}^D$ {Use the default configuration for unknown workloads.}

else

 if $\mathbb{C}_t.currentLabel$ has optimal config in WorkloadDB **then**

 $\mathfrak{J}_i^o \leftarrow$ get optimal config from WorkloadDB

else

 if $\mathbb{C}_t.currentLabel$ has workload drift **then**

 $\mathfrak{J}_i^o \leftarrow$ Explorer.localSearch($\mathfrak{J}_i$) {Do a local search starting with the last good configuration.}

else

 $\mathfrak{J}_i^o \leftarrow$ Explorer.globalSearch()

 end if

 Update WorkloadDB with $\mathfrak{J}_i^o$

 end if
end if

9.7 The KERMIT Off-Line Sub-System Architecture

Figure 9.5 shows the high-level processing pipeline flow in the off-line sub-system. The high-level component KERMIT workload analyser (KWanl in Figure 9.3) implements the off-line analytic pipeline which performs the following stages of processing:

1. Workload discovery and labelling.
2. Workload characterization.
3. Workload anticipation.
4. Training set generation.
5. Classifier training.

Sections below begin by discussing processing pipeline and the algorithm used to detect and identify new workloads. The next section builds on this discussion to describe the algorithm for detecting workload drift.

Figure 9.5: The high-level steps in the off-line sub-system processing pipeline.

9.7.1 Workload Discovery, Characterization, and Drift Detection

As discussed in the sections above, the KERMIT workload monitor stores the aggregated stream of workload windows $\{O_t\}_{t=1}^n$ in the transformation zone of the KERMIT knowledge base (see Figure 9.4). The high-level algorithm for workload discovery, characterization and drift detection is given in Algorithm 2.

The algorithm begins by using the ChangeDetector component in batch mode to scan the persisted time series of observation window data, and flag workload transition windows as described in Chapter 7. The logic of the batch operation is exactly the same as in the real-time use case. The workload transition windows are then removed from the original set into a separate set, and clustering analysis is performed on the now filtered set of workload observation windows.

Figure 9.6 shows the key performance metrics for several different clustering algorithms. The effectiveness of each algorithm was evaluated using time-series workload data recorded during the execution of Apache Hadoop and Spark benchmarks. Clustering results were compared to ground truth interpretation made by a human

Algorithm 2 The workload discovery and drift detection algorithm.

Input: $\{O_t\}_{t=1}^n \neq \emptyset$ {Landed observation window time-series.}

Output: $\{\mathcal{Y}\}_j^t$ {Set of identified workload labels.}

run ChangeDetector.batch() to identify transition windows

extract transition windows from $\{O_t\}_{t=1}^n$

run DBSCAN on $\{O_t\}_{t=1}^n$ to get a set of clusters

for all clusters in the set **do**

 calculate workload characterization statistics

if find match in WorkloadDB is True **then**

Y_j gets matched label from WorkloadDB

if L2 norm between the mean vectors of workload characterisations differ by more than ϵ **then**

 update isDrifting to True in WorkloadDB

 update workload characterisation for matching label in WorkloadDB with new data

end if

else

 generate new label for the new workload

 insert new workload label and characterization data into WorkloadDB

end if

end for

specialist using Apache Hadoop and Spark logs.

The key metrics indicated in the figure are Awt and Purity. Purity indicates how many of the observation windows were classified correctly by the on-line sub-system. The Awt metric is also an accuracy-type metric. It measures how accurately the algorithm was able to identify different workload types. For example, if the benchmark executed 3 different workload types and the algorithm detected 3 clusters whose centroids call within the observation window range of each workload type, then the Awt metric for this algorithm would be 100%.

Workload discovery in KERMIT is accomplished by running the DBSCAN algorithm on filtered observation window data. DBSCAN identifies clusters within the observation window data. Each cluster represents a distinct workload type.

The next step (see Algorithm 2) is to check if the newly identified workloads have been encountered before by calculating the workload characterization statistics for each cluster, and comparing with workload characterization statistics for workloads already identified and stored in WorkloadDB. This is accomplished by using the

Figure 9.6: Workload discovery performance for clustering algorithms.

ChangeDetector off-line to compare statistical data.

Workload characterization involves calculating the relevant statistics for each subset of observation windows that were grouped by the DBSCAN algorithm into the corresponding cluster. A full set of statistics, including the mean, the standard deviation, the max, the min, the 90th percentile, and the 75th percentile are calculated. This set of statistics is the workload characterization.

If a matching workload characterization is found in WorkloadDB (as identified by the ChangeDetector), then this is an existing workload. All of the windows in the cluster get tagged with the matched workload label from WorkloadDB.

The next step is to check the workload for drift. This is accomplished by calculating the L2 norm of the distance between the mean vectors of the new cluster and the one stored in WorkloadDB. If the difference is larger than the configurable hyperparameter ϵ , then drift has occurred, and the WorkloadDB is updated with the workload characterization for the new cluster.

If no matching workload is found in the WorkloadDB, then the KERMIT workload analyser generates a unique integer label for the cluster. The generated workload labels do not need to be human-legible. They just need to be unique to each identified cluster of observation windows. Currently KERMIT implements a simple integer counter, because this facilitates the generation of libsvm files for model training. Then new label, along with the workload characterization, is inserted into WorkloadDB.

Figure 9.7: Entity-relationship model of the WorkloadDB schema.

Figure 9.7 shows the WorkloadDB data model. Each workload is uniquely identified by its unique automatically generated label. Each workload contains the workload characterization statistics, a true/false field indicating whether the optimal configuration has been found, and a true/false field indicating whether workload drift has been detected. Each workload can have one configuration stored in the WorkloadDB. This configuration may or may not be the optimal configuration.

When a workload is initially identified, it will not have a configuration associated with it. This is because the configuration search is performed in real time by the on-line sub-system. Once the KERMIT plug-in performs the global search for the workload, it will update the WorkloadDB with the optimal configuration, and set the field indicating whether the optimal configuration has been found to True.

The next time clustering analysis is performed this occurs on a set interval, and on a new set of $\{O_t\}_{t=n+1}^{n+k}$ data collected during the last interval, where k is a constant hyper-parameter that controls the length or the batch used for clustering analysis. The entire process described in Algorithm 2 is repeated.

This approach allows KERMIT to continuously learn new workloads. New workloads are added to the WorkloadDB as they are detected during the off-line batch processing. Already known workloads are protected against workload drift because

their characterizations are regularly updated. Workloads are never deleted from WorkloadDB. Thus KERMIT retains a long-term memory of workloads, and the ability of the KERMIT on-line sub-system to recognize workloads improves over time.

9.7.2 Automated Classifier Training

The KERMIT on-line analytic pipeline includes several classifiers described in Chapter 6, Chapter 7, [12], and [11]. Paragraphs below provide a quick overview of their function and purpose, discuss their training requirements and describe how the process is automated.

The KERMIT on-line classification pipeline uses the following classifiers:

- *ChangeDetector*. This statistical classifier is a binary classifier that simply uses the Welch’s statistical test to distinguish steady state processing from workload transitions. This classifier does not require off-line training.
- *WorkloadClassifier*. This classifier is based on the random forest ensemble algorithm (see Chapter 4). This is a supervised classifier that does require off-line training.
- *TransitionClassifier*. This classifier is also based on the random forest ensemble algorithm (see Chapter 4). It also is a supervised classifier that does require off-line training.
- *ZSL Workload Classifier*. This component, described in Chapter 8 and in [14] re-uses the *WorkloadClassifier* class, and introduces the *WorkloadSynthesizer* component. This component needs to be trained off-line. It also generates synthetic class instances, which need to be merged into the training process for the *WorkloadClassifier*.
- *WorkloadPredictor*. This component is based on an LSTM neural network algorithm described in Chapter 4.

The training pipeline performs the following high-level steps (some of the steps performed as part of workload discovery are repeated for completeness):

1. Extract observation window range for each workload in WorkloadDB.

2. Use workload observation window id set to extract a set of analytic windows from the analytic window stream $\{A_t\}_{t=1}^n$ created by the Workload Monitor. For every observation window there is a matching analytic window. This becomes the training set for the WorkloadClassifier Dp_{Ω}^{Tr} .
3. Establish window ranges for workload transitions by scanning the sequence of analytic windows and marking ranges of windows that connect window sets that belong to each workload cluster, and correlating with workload transitions identified during the workload discovery phase.
4. Generate labels for each workload transition type. The same algorithm is used for label generation as that used for workloads. The labels don't need to be human-readable, just unique and consistent.
5. Transform the analytic window sequence $\{A_t\}_{t=1}^n$ to a rate of change sequence $\{A'_t\}_{t=1}^n$.
6. Extract transition windows from the transformed sequence - this forms the training set for the TransitionClassifier D_{Δ}^{Tr} .
7. Execute the WorkloadSynthesizer component on Dp_{Ω}^{Tr} to account for possible anticipated hybrid, multi-user workloads. This involves the following steps:
 - (a) Generating the Class Descriptor file described in Chapter 8 and in [14]. Each workload entry in the Workload DB is used as a pure class. Possible hybrid workloads are constructed by pairing the pure workloads.
 - (b) Generate labels for the anticipated, hybrid workloads using the same algorithm as for pure workload classes.
 - (c) Update the WorkloadDB with synthetic class prototypes - these contain the same information as the workload characterizations calculated for the seen classes.
 - (d) Merge the synthetic workload instances with the observed workload instances to construct the final, merged, WorkloadClassifier training set D_{Ω}^{Tr} .
8. Generate the training set for the WorkloadPredictor component D_{Ω}^{Tr} by extracting segments from the label sequence $\{Y_t\}_{t=n+1}^{n+k}$.

9. Train the classifiers.

Most of the steps described above, with the exception of steps needed to train the `WorkloadClassifier`, can be executed in parallel given sufficient compute resources.

9.8 Conclusion

This chapter presents the first architecture intended for autonomic optimization of big data workloads. The KERMIT architecture implements an autonomic feedback loop that includes on-line and off-line processing stages. It uses machine learning pervasively to analyse workload characteristics, identify new workload types, detect changes in real-time, classify workloads, and predict future workload types and characteristics. Change detection, workload classification, workload prediction, and parameter search are performed on-line, in real time. Classifier training is performed off-line as a batch machine learning pipe-line.

Experimental investigations focused on the critical proof points of the autonomic feedback loop demonstrate that the KERMIT architecture can:

- Real-time workload classification with 90% accuracy [11].
- Detect workload changes in real-time with 99% accuracy [11].
- Predict workload type with up to 96% accuracy [11].
- Anticipate new, unseen workload types, and classify them with 83% accuracy [14].

The KERMIT architecture, as discussed above, can anticipate the appearance of new unseen, multi-user, hybrid workloads that can present a mix of characteristics observed with the currently identified workloads. This is a capability that has not been previously described in any of the earlier works focused on small data and cloud workload segments.

The ability to adjust to workload drift is another key element of the KERMIT architecture. The on-line sub-system will do its best to classify workloads in real-time. If a previously unseen and unanticipated workload is encountered, the KERMIT on-line sub-system will initially classify it as one of the known workload types, with the closest characteristics, and use the best available configuration for that workload. This is often better, in terms of reducing the tuning overhead, than immediately

performing a global search for that workload. The new workload will be discovered by the off-line sub-system the next time it performs clustering analysis.

This architecture can operate with minimal configuration by a human administrator. Although there are still a number hyper-parameters that need to be set, these, unlike many of the Apache Hadoop and Spark configuration settings, do not require frequent tuning. For the most part these hyper-parameters can be left at their default settings. These default settings (for example the μ hyper-parameter for the DBSCAN algorithm) in many cases apply to a broad range of conditions and are well-documented in the scientific and technical literature.

Chapter 10

Conclusion

10.1 Summary

The research studies presented as part of this thesis demonstrate that it is possible to develop sophisticated on-line autonomic systems for the big data stack. They demonstrate that it is possible to tune the container density on-line, achieving performance results equal to, or better than, rule-of-thumb tuning typically used by a human practitioner. Although this was demonstrated using the Hadoop MapReduce, Apache Spark and YARN technologies - the architectural analysis presented in Chapter 2 shows that these findings would be generally applicable, and extensible, to the other frameworks and resource managers as their level of technical maturity grows and more framework-to-resource manager integrations are developed. Findings from this study also showed that optimizing the frequency of the parameter space searches is key to achieving the ideal performance improvements.

Optimizing the frequency of the parameter searches is, by itself, a fairly complex problem that requires us to be able to classify workloads, accurately identify and classify changes in workload characteristics, and predict future workload transitions. Studies presented in Chapter 6 and Chapter 7 demonstrate that it is possible to leverage machine learning algorithms to accomplish this with high degree of accuracy.

Research studies presented in this thesis include a number of firsts:

- KERMIT is the first autonomic architecture specifically designed for the big data stack.
- KERMIT incorporates the first automatic tuning engine capable of tuning both Apache Hadoop and YARN.

- KERMIT is the first system to demonstrate the ability to achieve the same, or better, performance gains as typical tuning by a human practitioner. Other works compare performance gains against the default settings.
- KERMIT is the first system to demonstrate significant performance gains on Terabyte-scale data sets. The other studies were performed on much smaller, Gigabyte-scale, data sets. Scale matters in big data space because it introduces bottlenecks that can fundamentally alter the workload characteristics - e.g. from CPU to I/O bound processing.
- KERMIT is the first autonomic system to be able to anticipate and classify previously unseen workloads.

KERMIT includes the following new algorithms designed specifically to support automatic tuning and autonomic systems:

1. The Explorer low-overhead search algorithm.
2. The real-time, statistical ensemble, change detection algorithm for multi-variate, continuous data.
3. The zero-shot algorithm for classification of hybrid classes.

This thesis also introduces a number of novel concepts related to workload analysis and optimization, and autonomic computing:

1. **The definition of workload.** This thesis defines the term workload in a narrower, but a very clear mathematical manner. This is different from definitions used by other researchers. This approach ultimately enables more complex analysis.
2. **Workload transitions.** This is a new concept that describes non-steady-state of processing.
3. **Workload grammar.** Workloads and transitions can be classified and transformed into a sequence of labels that can be processed in a manner similar to natural languages.
4. **Workload drift.** This concept refers to non-specific changes that may occur in workload characteristics over time.

5. **Workload-aware automatic tuning.** This thesis, unlike previous works, introduces the notion that automatic tuning engines should be aware of the workload they are trying to optimize - similar to an experienced human practitioner's approach.

The implications of these innovations for the broader big data technology stack are discussed in the section below.

10.2 Implications for the Big Data Stack Architecture

Findings presented in this thesis make possible an architectural integration style shown in Figure 10.1. KERMIT analyses, classifies, and characterises workloads. It produces a workload context that contains, as discussed in Chapter 9, the workload characterisation, the optimal configuration that should be used, and predicted workload types and their associated contexts for several future prediction horizons.

The resource manager invokes the scheduler in response to resource requests arriving from the analytic frameworks such as Apache Hadoop MapReduce or Spark. The scheduler allocates resources based on the underlying algorithm that it implements. The schedulers available today, such as the fair scheduler or the capacity scheduler, are reactive in nature. This means that, when it receives a request for resources from the resource manager, it will allocate all available resources to it based on the configured policy. When a new request arrives it can either wait until some of the tasks of the previous requests have completed and released the associated resources, or it can pre-empt some of the currently executing tasks and restart them at a later time.

However, how do we know how many tasks to pre-empt, as opposed to just wait for them to complete? This is an optimization problem. Currently, it's a tuning problem. The administrator has to set configuration parameters that control pre-emption.

KERMIT opens the door to the creation of more intelligent scheduling and pre-emption algorithms that can leverage the predictions contained in the workload context. If the scheduler knows what types workloads to expect in the future, and their characteristics, it can make more intelligent decisions about pooling resources and whether to pre-empt currently executing tasks or not.

Figure 10.1: Possible integration among KERMIT, the resource manager and the analytic frameworks.

Consider the following example. The currently executing workload is the map phase of the Hadoop MapReduce processing. The prediction indicates that there will be a workload transition from the CPU-hungry map processing to the I/O bound, and much less CPU-hungry, shuffle processing within 5 observation windows. Now a new request for resources arrives. The scheduler, knowing that CPU and memory resources are likely to free up soon, could decide not to pre-empt tasks that are already executing, and simply wait for the currently executing map tasks to finish. Alternatively, if the prediction indicates that the map processing is likely to continue longer, the scheduler could decide to immediately pre-empt some of the map tasks to make room for the new job.

The analytic frameworks, Such as Apache Spark and Tez, could also leverage the workload context generated by KERMIT. These frameworks implement two-level scheduling as discussed in Chapter 2. The central schedule (in our case YARN) performs the first level of coarse resource allocation and then the analytic framework performs the second level of scheduling to assign resources assigned to it during the first level to tasks. These frameworks could also use the KERMIT-generated workload context to improve their scheduling algorithms. The additional performance improvements gained from optimizing both levels of scheduling could be significant.

Figure 10.2 how the Hadoop technology stack can be extended with an Autonomic

Figure 10.2: KERMIT positioning within the logical big data stack.

Optimization Layer - implemented in this thesis as KERMIT. This new layer would interact with the Hardware/IaaS layers to collect resource utilisation metrics. Although this was not attempted during this thesis due to existing HDFS capabilities, this distributed file system could potentially be extended to provide additional helpful information to KERMIT, and to possibly use the KERMIT-generated workload context to optimize data placement.

As discussed above, KERMIT integrates with the Resource Management layer, and could be used by the analytic framework layer as well. KERMIT could be integrated with the monitoring and management layer to provide human-readable graphics and reports summarizing workload-related statistics and autonomic optimization efficiency. Ingestion frameworks, such as Sqoop, are typically implemented on frameworks such as Apache Spark and could also benefit from KERMIT-generated workload context.

The section below recaps possible avenues for future work.

10.3 Future Work

Research presented in this thesis opens up new possibilities for further intelligent workload optimization. The algorithms presented in the preceding chapters can also be adapted to other computer science domains, and thus proposed future research can be organized around the following long-term goals:

1. Develop new adaptive scheduling algorithms for optimizing container density and resource allocation under rapidly changing workload conditions.
2. Develop new adaptive pre-emption algorithms for reclaiming and optimizing resource allocation and sharing among hosted applications.
3. Develop new advanced on-line machine learning algorithms to support objectives 1 and 2 and adapt them for general use in other domains.

In order to enable long-term goals the following short-term objectives need to be achieved:

1. Propose new benchmarks, or enhancements to existing benchmarks, as necessary, needed to evaluate the cloud container orchestration performance under highly dynamic, non-steady-state, workload conditions.
2. Develop new real-time machine learning algorithms capable of learning on-line, without requiring an off-line training phase.

The first short-term objective is necessary because there are currently very few benchmarks suitable for analysing software performance under non-steady-state conditions. The second short term objective is necessary because the currently popular supervised and unsupervised machine learning algorithms are either meant for off-line use cases, and/or require an expensive off-line training phase. While this thesis demonstrates that it is possible to build an autonomic closed feedback loop using a combination of on-line and off-line techniques it would be preferable to leverage algorithms and techniques capable of learning on-line.

The starting point should be to define the theoretical base, and evaluation metrics, needed to analyse and compare applications and systems during non-steady state workload conditions. This should be followed by a proof-of-concept aimed at developing new adaptive scheduling and pre-emption algorithms. The proof of concept will include benchmark development work, and experimental evaluation of the new algorithm performance relative to common existing scheduling algorithms.

Benchmark development work, as stated above, is necessary because currently available benchmarks focus almost exclusively on evaluating steady-state performance of applications and systems. Few truly stress the container orchestration layer. Once new scheduling and pre-emption algorithms and suitable benchmarks are ready, they

should be evaluated head-to-head against each other and against the current state-of-the-art on the same cloud environment.

Another potential avenue for future research would be to apply the machine learning algorithms developed during this thesis and intended primarily for workload classification and optimization to other domains. The change detection algorithm could be attempted with other types of streaming data, such as telecommunication data and data generated by networks of sensors. The zero-shot technique for identifying unseen hybrid classes could be broadly applicable to many domains including document and image classification.

List of References

- [1] Ziad Al-Halah and Rainer Stiefelhagen. How to transfer? zero-shot object recognition via hierarchical transfer of semantic attributes. In *IEEE Winter Conference on Applications of Computer Vision (WACV'15)*, pages 837 – 843. IEEE, 2015.
- [2] Ahsan Javed Awan, Mats Brorsson, Vladimir Vlassov, and Eduard Ayguade. Micro-architectural characterization of apache spark on batch and stream processing workloads. In *Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom)(BDCloud-SocialCom-SustainCom), 2016 IEEE International Conferences on*, pages 59–66. IEEE, 2016.
- [3] Babak Behzad, Huong Vu Thanh Luu, Joseph Huchette, Surendra Byna, Ruth Aydt, Quincey Koziol, Marc Snir, et al. Taming parallel i/o complexity with auto-tuning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 68. ACM, 2013.
- [4] Michael J Cafarella and Christopher Ré. Manimal: relational optimization for data-intensive programs. In *Proceedings of the 13th International Workshop on the Web and Databases*, page 10. ACM, 2010.
- [5] David Carrera, Malgorzata Steinder, Ian Whalley, Jordi Torres, and Eduard Ayguade. Autonomic placement of mixed batch and transactional workloads. *IEEE Transactions on Parallel and Distributed Systems*, 23(2):219 – 231, 2012.
- [6] Quan Chen, Minyi Guo, Qianni Deng, Long Zheng, Song Guo, and Yao Shen. Hat: history-based auto-tuning mapreduce in heterogeneous environments. *The Journal of Supercomputing*, 64(3):1038–1054, 2013.
- [7] Ludmila Cherkasova, Kivanc Ozonat, Ninfang Mi, Julie Symons, and Evgenia Smirni. Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 452 – 461. IEEE, 2014.
- [8] Xiaoan Ding, Yi Liu, and Depei Qian. Jellyfish: Online performance tuning with adaptive configuration and elastic container in hadoop yarn. In *Parallel and Distributed Systems (ICPADS), 2015 IEEE 21st International Conference on*, pages 831–836. IEEE, 2015.

- [9] Docker. Docker architecture. Visited on 2019-01-29. URL: <https://docs.docker.com/engine/docker-overview/>.
- [10] Said Elnaffar and Patrick Martin. The psychic-skeptic prediction framework for effective monitoring of dbms workloads. *Data & Knowledge Engineering*, 68:393–414, 2009.
- [11] M. Genkin and F. Dehne. Autonomic workload change classification and prediction for big data workloads. In *2019 IEEE International Conference on Big Data 2019 (Big Data)*, Los Angeles, CA, USA., pages 2835 – 2844. IEEE, 2019.
- [12] M. Genkin, F. Dehne, Pablo Navarro, and Siyu Zhou. Machine-learning based spark and hadoop workload classification using container performance patterns. In C. Zheng and J. Zhan (Eds) *2018 International Symposium on Benchmarking, Measuring and Optimizing (Bench’ 18)*, volume 11459, pages 118 – 130. Springer, 2019.
- [13] M. Genkin, F. Dehne, M. Pospelova, Y. Chen, and P. Navarro. Automatic, on-line tuning of yarn container memory and cpu parameters. In *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems*, pages 317–324. IEEE, 2016.
- [14] Mikhail Genkin. Zero-shot machine learning technique for classification of multi-user big data workloads. In *2020 IEEE International Conference on Big Data (Big Data)*, Atlanta, GA, USA., page in print. IEEE, 2020.
- [15] Ian Gergin, Bradley Simmons, and Martin Litoiu. A decentralized autonomic architecture for performance control in the cloud. In *2014 IEEE International Conference on Cloud Engineering, Boston, MA*, pages 574 – 579. IEEE, 2014.
- [16] Klaus Greff, Rupesh K. Srivastava, Jan Koutnik, Ras R. Steunebrink, and Jurgen Schmidhuber. Lstm: A search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, 28(10):2222 – 2232, 2016.
- [17] Apache Hadoop. Apache hadoop yarn architecture. Visited on 2019-01-29. URL: <https://hadoop.apache.org/docs/r3.1.2/hadoop-yarn/hadoop-yarn-site/YARN.html>.
- [18] Herodotos Herodotou and Shivnath Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *Proc. of the VLDB Endowment*, 4(11):1111–1122, 2011.
- [19] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A self-tuning system for big data analytics. In *CIDR*, volume 11, pages 261–272, 2011.
- [20] Tin Kam Ho. The random subspace method for constructing decision forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8):832 – 844, 1998.

- [21] Marc Holtze and Norbert Ritter. Autonomic databases: Detection of workload shifts with n-gram-models. In *ADBIS 2008: Advances in Databases and Information Systems*, volume 5207 of *Lecture Notes in Computer Science*, pages 127 – 142. Springer, 2008.
- [22] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pages 41–51. IEEE, 2010.
- [23] Zheng Huang, Jiajun Peng, Huijuan Lian, Jie Guo, and Weidong Qiu. Deep recurrent model for server load and performance prediction in data center. *Hindawi Complexity*, 2017, 2017.
- [24] IBM. An architectural blueprint for autonomic computing. *IBM*, 7:1 – 31, 2005.
- [25] Impetus. Hadoop performance tuning, 2012.
- [26] IntelliPaat. Apache hadoop ecosystem. Visited on 2015-07-30. URL: <https://cdn.intellipaat.com/mediaFiles/2015/07/Apache-Hadoop-Ecosystem.png>.
- [27] Zhen Jia, Chao Xue, Guancheng Chen, Jianfeng Zhan, Lixin Zhang, Yonghua Lin, and Peter Hofstee. Auto-tuning spark big data workloads on power8: Prediction-based dynamic smt threading. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, pages 387–400. ACM, 2016.
- [28] Zhen Jia, Jianfeng Zhan, Lei Wang, Rui Han, Sally A McKee, Qiang Yang, Chunjie Luo, and Jingwei Li. Characterizing and subsetting big data workloads. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 191–201. IEEE, 2014.
- [29] Rahul Khanna, Mrityika Ganguli, Ananth Narayan, Abhiram R., and Piyush Gupta. Autonomic characterization of workloads using workload fingerprinting. In *Proceedings of 2014 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*. IEEE, 2014.
- [30] Kubernetes. Apache spark with native kubernetes support. Visited on 2018-03-08. URL: <https://kubernetes.io/blog/2018/03/apache-spark-23-with-native-kubernetes/>.
- [31] Christoph H. Lampert, Hannes Nickisch, and Stefan Harmeling. Learning to detect unseen object classes by between-class attribute transfer. In *Proceedings: IEEE Conference on Computer Vision and Pattern Recognition (CVPR'09)*, pages 951 – 958. IEEE, 2019.
- [32] Zhou Lei, Bolin Hu, Jianhua Guo, Loukai Hu, Wenfeng Shen, and Yu Lei. Scalable and efficient workload hotspot detection in virtualized environment. *Cluster Computing*, 17:1253–1264, 2014.
- [33] Min Li, Liangzhao Zeng, Shicong Meng, Jian Tan, Li Zhang, Ali R Butt, and Nicholas Fuller. Mronline: Mapreduce online performance tuning. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 165–176. ACM, 2014.

- [34] Guangdeng Liao, Kushal Datta, and Theodore L Willke. Gunther: search-based auto-tuning of mapreduce. In *Euro-Par 2013 Parallel Processing*, pages 406–419. Springer, 2013.
- [35] Todd Lipcon. Cloudera: 7 tips for improving mapreduce performance, 2009.
- [36] Todd Lipcon. Cloudera: Optimizing mapreduce job performance, 2012.
- [37] Edson Ramiro Lucas Filho, Eduardo Cunha De Almeida, Yves Le Traon, et al. Intra-query adaptivity for mapreduce query processing systems. In *IDEAS 2014: 18th International Database Engineering & Applications Symposium*, 2014.
- [38] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. Query-based workload forecasting for self-driving database management systems. In *Proceedings of 2018 International Conference on Management of Data (SIGMOD’18)*. ACM, 2018.
- [39] Asit K Mishra, Joseph L Hellerstein, Walfredo Cirne, and Chita R Das. Towards characterizing cloud backend workloads: insights from google compute clusters. *ACM SIGMETRICS Performance Evaluation Review*, 37(4):34–41, 2010.
- [40] Ismael Solis Moreno, Peter Garraghan, Paul Townend, and Jie Xu. An approach for characterizing workloads in google cloud to derive realistic resource utilization models. In *Service Oriented System Engineering (SOSE), 2013 IEEE 7th International Symposium on*, pages 49–60. IEEE, 2013.
- [41] Zeinab Movahedi, Mouna Ayari, Rami Langar, and Guy Pujolle. A survey of autonomic network architectures and evaluation criteria. *IEEE Communications Surveys & Tutorials*, 14(2):464 – 490, 2012.
- [42] Wira D Mulia, Naresh Sehgal, Sohumi Sohoni, John M Acken, C Lucas Stanberry, and David J Fritz. Cloud workload characterization. *IETE Technical Review*, 30(5):382–397, 2013.
- [43] Jinseok Nam, Eneldo Loza Mencía, Hyunwoo J. Kim, and Johannes Fürnkranz. Predicting unseen labels using label hierarchies in large-scale multi-label learning. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECMLPKDD’15)*, pages 102 – 118, 2015.
- [44] Seyed Mohammad Reza Nouri, Han Li, Srikumar Venugopal, Wenxia Guo, MingYun He, and Wenhong Tian. Autonomic decentralized elasticity based on a reinforcement learning controller for cloud applications. *Future Generation Computer Systems*, 94:765 – 780, 2019.
- [45] Bazit Raza, Asma Sher, Sana Afzal, Ahmad Kamran Malik, Adeel Anjum, Yogan Jaya Kumar, and Muhammad Faheem. Autonomic workload performance tuning in large-scale data repositories. *Knowledge and Information Systems*, 61(2):27 – 63, 2019.
- [46] LLC Read Evaluate Press. Dbscan. Visited on 2021-2-3. URL: <https://github.com/NSHipster/DBSCAN>.

- [47] David E Rumhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(10), 1986.
- [48] Christoph A Schaefer, Victor Pankrati, and Walter F Tichy. Atune-il: An instrumentation language for auto-tuning parallel applications. In *Euro-Par 2009 Parallel Processing*, pages 9–20. Springer, 2009.
- [49] Apache Spark. Spark cluster architecture. Visited on 2019-01-29. URL: <https://spark.apache.org/docs/latest/cluster-overview.html>.
- [50] 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 (HPCC/SmartCity/DSS), 2016 IEEE 18th International Conference on*, pages 586–593. IEEE, 2016.
- [51] Kun Wang, Ben Tan, Juwei Shi, and Bo Yang. Automatic task slots assignment in hadoop mapreduce. In *Proceedings of the 1st Workshop on Architectures and Systems for Big Data*, pages 24–29. ACM, 2011.
- [52] Wei Wang, Vincent W. Zheng, Han Yu, and Chunyan Miao. A survey of zero-shot learning: Settings, methods, and applications. *ACM Transactions on Intelligent Systems and Technology*, 10(2), 2019.
- [53] Md Wasi-Ur-Rahman, Nusrat Sharmin Islam, Xiaoyi Lu, Dipti Shankar, and Dhabaleswar K Panda. Mr-advisor: A comprehensive tuning tool for advising hpc users to accelerate mapreduce applications on supercomputers. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2016 28th International Symposium on*, pages 198–205. IEEE, 2016.
- [54] Wikipedia. Machine learning. Visited on 2020-09-20. URL: https://en.wikipedia.org/wiki/Machine_learning.
- [55] Wikipedia. Mean squared error. Visited on 2020-10-26. URL: https://en.wikipedia.org/wiki/Mean_squared_error.
- [56] Wikipedia. Receiver operating characteristic. Visited on 2020-11-09. URL: https://en.wikipedia.org/wiki/Receiver_operating_characteristic.
- [57] Nezih Yigitbasi, Theodore L Willke, Guangdeng Liao, and Dick Epema. Towards machine learning-based auto-tuning of mapreduce. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2013 IEEE 21st International Symposium on*, pages 11–20. IEEE, 2013.
- [58] Freund Yoav and Schapire Robert E. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119 – 139, 1997.
- [59] Rui Zhang, Min Li, and Dean Hildebrand. Finding the big data sweet spot: Towards automatically recommending configurations for hadoop clusters on docker containers. In *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, pages 365–368. IEEE, 2015.