Cost-efficient Management of Cloud Resources for Big Data Applications

Muhammed Tawfiqul Islam

Submitted in total fulfilment of the requirements of the degree of

Doctor of Philosophy

School of Computing and Information Systems
THE UNIVERSITY OF MELBOURNE

January 2021

ORCID: 0000-0003-4922-7807

Copyright © 2021 Muhammed Tawfiqul Islam

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm or any other means without written permission from the author.

Cost-efficient Management of Cloud Resources for Big Data Applications

Muhammed Tawfiqul Islam
Principal Supervisor: Prof. Rajkumar Buyya
Co-Supervisor: Prof. Shanika Karunasekera

Abstract

Analyzing a vast amount of business and user data on big data analytics frameworks is becoming a common practice in organizations to get a competitive advantage. These frameworks are usually deployed in a computing cluster to meet the analytics demands in every major domain, including business, government, financial markets, and health care. However, buying and maintaining a massive amount of on-premise resources is costly and difficult, especially for start-ups and small business organizations. Cloud computing provides infrastructure, platform, and software systems for storing and processing of data. Thus, Cloud resources can be utilized to set up a cluster with a required big data processing framework. However, several challenges need to be addressed for Cloud-based big data processing which includes: deciding how much Cloud resources are needed for each application, how to maximize the utilization of these resources to improve applications' performance, and how to reduce the monetary cost of resource usages. In this thesis, we focus on a user-centric view, where a user can be either an individual or a small/medium business organization who want to deploy a big data processing framework on the Cloud. We explore how resource management techniques can be tailored to various user-demands such as performance improvement, and deadline guarantee for the applications; all while reducing the monetary cost of using the cluster. In particular, we propose efficient resource allocation and scheduling mechanisms for Cloud-deployed Apache Spark clusters.

This thesis advances the state-of-the-art by making the following contributions:

1. A comprehensive literature review on the resource management of big data applications in Cloud computing environments. In addition, a taxonomy on the scheduling of big data applications in the Cloud environment while tackling vari-

ous user demands.

2. A mathematical model to estimate an application's completion time depending on various configuration parameters and workload characteristics, and a resource allocation mechanism to use the estimate for cost-efficient fine-grained Cloud resource allocation for big data applications.

3. Two dynamic job scheduling algorithms which can adapt to cluster resource changes, prioritize time-critical jobs and minimize the Virtual Machine (VM) usage cost of the whole cluster.

4. Two job scheduling algorithms which leverage the pricing model of Cloud VM instances to minimize VM usage cost in a big data computing cluster deployed on hybrid-cloud.

5. A Reinforcement Learning (RL) model of the job scheduling problem, reward formulation for multiple-objectives, and Deep Reinforcement Learning (DRL) based job scheduling agents which can automatically learn inherent cluster and job characteristics to optimize multiple objectives.

Declaration

This is to certify that

1. the thesis comprises only my original work towards the PhD,
2. due acknowledgement has been made in the text to all other material used,
3. the thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Muhammed Tawfiqul Islam, January 2021

Preface

This thesis research has been carried out in the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne. The main contributions of the thesis are discussed in Chapters 2-6 and are based on the following publications:

- **Muhammed Tawfiqul Islam** and Rajkumar Buyya, "Resource Management and Scheduling for Big Data Applications in Cloud Computing Environments", *Handbook of Research on Cloud Computing and Big Data Applications in IoT*, B. Gupta and D. Agrawal (eds), 1-23pp, ISBN-13: 978-1522584070, IGI Global, USA, 2019.

- **Muhammed Tawfiqul Islam**, Shanika Karunasekera, and Rajkumar Buyya, "dSpark: Deadline-based Resource Allocation for Big Data Applications in Apache Spark", Proceedings of the 13th IEEE International Conference on e-Science (e-Science 2017), Auckland, New Zealand, October 24-27, 2017.

- **Muhammed Tawfiqul Islam**, Satish N. Srirama, Shanika Karunasekera, and Rajkumar Buyya, "Cost-efficient Dynamic Scheduling of Big Data Applications in Apache Spark on Cloud", *Journal of Systems and Software (JSS)*, Volume 162, Pages: 1-14, ISSN: 0164-1212, Elsevier Press, Amsterdam, The Netherlands, April 2020.

- **Muhammed Tawfiqul Islam**, Huaming Wu, Shanika Karunasekera, and Rajkumar Buyya, "SLA-based Scheduling of Spark Jobs in Hybrid Cloud Computing Environments", *IEEE Transactions on Computers (TC)* [Under 2nd Revision].

- **Muhammed Tawfiqul Islam**, Shanika Karunasekera, and Rajkumar Buyya, "Cost and Performance-oriented Spark Job Scheduling in Cloud with Deep Reinforcement Learning", *IEEE Transactions on Parallel and Distributed Systems (TPDS)* [Under Review].

Acknowledgements

All praise to Almighty ALLAH for blessing me with the strength and courage to complete my PhD studies and I convey my humblest gratitude to the Holy Prophet Muhammad (Peace be upon him).

I would like to thank my supervisors, Professor Rajkumar Buyya and Professor Shanika Karunasekera, for the opportunity to pursue my PhD. I am truly grateful for their invaluable supervision, support, and motivation throughout my candidature.

I would also like to express my gratitude to the chair of my PhD advisory committee, A/Prof. Michael Kirley, for his advice. I am thankful to Dr Xunyun Liu and Dr Minxian Xu for their constructive comments and technical advice at the beginning of my PhD journey.

Special thanks to my undergraduate and graduate thesis supervisor, Dr Syed Faisal Hasan for always encouraging me to be a better researcher. I would also like to thank all the past and current members of the CLOUDS Laboratory, at the University of Melbourne. In particular, I thank Dr Satish Narayana Srirama, Dr Huaming Wu, Dr Adel Nadjaran Toosi, Dr Md. Redowan Mahmud, Dr Chenhao Qu, Dr Maria Rodriguez, Dr Jungmin Jay Son, Dr Yaser Mansouri, Dr Safiollah Heidari, Dr Sara Kardani, Shashikant Ilager, Md. H. Hilman, TianZhang He, Mohammad Goudarzi, Zhiheng Zhong, Samodha Pallewatta, Amanda Jayanetti, Anupama Mampage, and Jie Zhao for their support.

I acknowledge the University of Melbourne and the Australian Research Council (Discovery Project) for granting me with scholarships to pursue my doctoral studies.

I would like to express my gratitude to my parents Tanvir Jahan and Jafar Ahmed, for their endless support and motivation throughout the PhD journey. I would also like to thank my siblings Dr Tahmina Akhter, Taslima Akhter, and Dr Muhammed Kamrul Islam for always assisting me in every aspect of my life.

Last but not least, I am thankful to my wife Tahrima Hashem for her unconditional love, appreciations and understandings. It was a pleasure to be her PhD colleague, which helped both of us to grow and support each other.

Muhammed Tawfiqul Islam
Melbourne, Australia
January 2021

Contents

List of Figures **xv**

List of Tables **xviii**

1 Introduction **1**
1.1 Challenges in Cost-efficient Resource Management 3
1.1.1 Challenges in Resource Allocation 3
1.1.2 Challenges in Dynamic Job Scheduling 4
1.1.3 Challenges in Job Scheduling in Hybrid Cloud 4
1.1.4 Challenges in Intelligent Learning of Job Scheduling Algorithms . 5
1.2 Research Problems and Objectives 5
1.3 Evaluation Methodology 7
1.4 Thesis Contributions 7
1.5 Thesis Organization 9

2 A Taxonomy and Systematic Review **13**
2.1 Introduction 13
2.2 Background 15
2.2.1 Cloud Computing 15
2.2.2 Big Data 16
2.2.3 Big Data Processing Frameworks 17
2.2.4 Cluster Managers 21
2.3 Resource Management for Big Data Applications 24
2.3.1 Setup Layer 24
2.3.2 Operation Layer 26
2.3.3 Maintenance Layer 29
2.4 A Taxonomy on Scheduling of Big Data Applications 31
2.4.1 Objective 32
2.4.2 Approach 34
2.4.3 Architecture 35
2.4.4 Framework 35
2.5 Summary 37

3 Deadline-based Cloud Resource Allocation for Big Data Applications 39
3.1 Introduction . . . 39
3.2 Background . . . 42
3.3 Related Work . . . 43
3.4 Problem Formulation . . . 46
3.4.1 Cost-efficient Resource Allocation Model . . . 46
3.4.2 Application Completion Time Prediction Model . . . 48
3.5 dSpark Framework Overview . . . 50
3.6 Performance Evaluation . . . 55
3.6.1 Implementation . . . 55
3.6.2 Experimental Setup . . . 55
3.6.3 Analysis of Results . . . 58
3.7 Summary . . . 61

4 Scheduling Big Data Applications in Cloud Computing Environments 63
4.1 Introduction . . . 63
4.2 Background . . . 66
4.2.1 Apache Spark . . . 66
4.2.2 Apache Mesos . . . 68
4.3 Related Work . . . 69
4.4 Cost-efficient Job Scheduling . . . 72
4.4.1 Motivation . . . 72
4.4.2 Problem Formulation . . . 76
4.4.3 Job Scheduler . . . 78
4.4.4 Executor Placement . . . 82
4.4.5 Complexity Analysis . . . 84
4.5 System Design and Implementation . . . 85
4.6 Performance Evaluation . . . 87
4.6.1 Experimental Setup . . . 87
4.6.2 Evaluation of Cost Efficiency . . . 90
4.6.3 Evaluation of Job Performance . . . 94
4.6.4 Evaluation of Deadline Violation . . . 94
4.6.5 Evaluation of Scheduling Overhead . . . 95
4.6.6 Effects of Resource Unification Thresholds (RUT) . . . 96
4.6.7 Discussion . . . 97
4.7 Summary . . . 98

5 Scheduling Big Data Applications in Hybrid Cloud 99
5.1 Introduction . . . 99
5.2 Background . . . 102
5.2.1 Apache Spark . . . 102
5.2.2 Apache Mesos . . . 103
5.2.3 Scheduling Levels . . . 104
5.2.4 Hybrid Cloud Deployment . . . 105

5.3 Related Work . . . 105
5.4 SLA-based Job Scheduling . . . 108
5.4.1 System Model . . . 108
5.4.2 Local Resource Model . . . 112
5.4.3 Cloud Resource Model . . . 113
5.4.4 Problem Formulation . . . 115
5.5 Proposed Job Scheduling Algorithms . . . 116
5.5.1 First Fit (FF) Heuristic-based Algorithm . . . 117
5.5.2 Greedy Iterative Optimization (GIO) Algorithm . . . 118
5.5.3 Complexity Analysis . . . 120
5.6 Performance Evaluation - Simulation . . . 120
5.6.1 Simulation Setup . . . 121
5.6.2 Baseline Schedulers . . . 122
5.6.3 Simulation Results . . . 123
5.7 Performance Evaluation - Real Experiments . . . 131
5.7.1 System Implementation . . . 131
5.7.2 Real Experiment Setup . . . 133
5.7.3 Benchmarking Applications . . . 134
5.7.4 Real Experiment Results . . . 136
5.8 Summary . . . 138

6 Learning Scheduling Algorithms with Deep Reinforcement Learning (DRL) 141
6.1 Introduction . . . 141
6.2 Related Work . . . 144
6.2.1 Framework Schedulers . . . 144
6.2.2 Performance model and Heuristic-based Schedulers . . . 145
6.2.3 DRL-based Schedulers . . . 146
6.3 Problem Formulation . . . 148
6.4 RL Model . . . 151
6.5 DeepRL Agents for Job Scheduling . . . 158
6.5.1 DQN Agent . . . 158
6.5.2 REINFORCE Agent . . . 160
6.6 RL Environment Implementation . . . 162
6.7 Performance Evaluation . . . 164
6.7.1 Experimental Settings . . . 164
6.7.2 Convergence of the DRL Agents . . . 167
6.7.3 Learning Resource Constraints . . . 170
6.7.4 Evaluation of Cost-efficiency . . . 170
6.7.5 Evaluation of Job Duration . . . 172
6.7.6 Evaluation of Multiple Reward Maximization . . . 177
6.7.7 Learned Strategies . . . 177
6.8 Summary . . . 178

7 Conclusions and Future Directions **179**
7.1 Summary of Contributions . . . 179
7.2 Future Research Directions . . . 181
7.2.1 Intelligent Resource Management . . . 181
7.2.2 Application Performance Modelling in Heterogeneous Resources . 181
7.2.3 Energy-efficient Management of Big Data Processing Systems . . . 182
7.2.4 AI-enhanced Autonomous Resource Selection and Allocation . . . 182
7.2.5 Straggler Detection and Mitigation in Job Scheduling . . . 182
7.2.6 Resource Management in Multi-tier Edge and Cloud Deployed Cluster . . . 183
7.2.7 Improving Energy Efficiency . . . 183
7.2.8 Scheduling Jobs in Geo-distributed Big Data Clusters . . . 183
7.2.9 Shared-sensing in Internet of Things (IoT) . . . 184
7.2.10 Fault Tolerant Resource Management . . . 184
7.2.11 Integrating Multiple Big Data Platforms . . . 184
7.3 Final Remarks . . . 185

List of Figures

1.1 Thesis Organization . . . 10

2.1 Big Data 3V . . . 17
2.2 A Taxonomy of Big Data Processing Frameworks . . . 18
2.3 Apache Hadoop Architecture . . . 19
2.4 Apache Spark Architecture . . . 20
2.5 Apache Hadoop Yarn Architecture . . . 22
2.6 Apache Mesos Architecture . . . 23
2.7 Key components of Resource Management in a Big Data Cluster . . . 25
2.8 A Taxonomy of Scheduling of Big Data Applications on Cloud . . . 31

3.1 An Apache Spark Cluster . . . 43
3.2 dSpark Architecture . . . 51
3.3 Application completion time modelling. . . . 53
3.4 Accuracy of application completion time prediction for different applications. . . . 54
3.5 Cost of resource usages by the proposed and the default approach for different applications. . . . 57
3.6 Comparison of resource usages between the proposed and the default approach . . . 59

4.1 Job submission frequencies in a single day (Facebook Hadoop Workload Trace-2009) . . . 73
4.2 An example cluster with different types of Jobs and VMs . . . 74
4.3 Different Executor Placement Strategies . . . 74
4.4 The implementation of the prototype system on top of Apache Mesos . . . 86
4.5 Cost comparison between the scheduling algorithms under different workload types . . . 91
4.6 Comparison between the scheduling algorithms regarding average job completion times under different workload types . . . 93
4.7 Comparison of deadline violations by different scheduling algorithms . . . 93
4.8 Effects of Resource Unification Threshold (RUT) values on average job completion time and cost . . . 97

5.1 Proposed Hybrid Cloud Model. The resource managers (cloud and local) are controlled by the scheduler to create executors of job in VMs, turn on/off VMs, and to monitor the cluster states. 111
5.2 Cost comparison between the scheduling algorithms under different VM instance pricing models in a lightly loaded cluster (the lower the better). The scheduling delay is omitted to show how close the schedulers perform to the MILP solution regarding true cost calculation. 124
5.3 Cost comparison between the scheduling algorithms under different VM instance pricing models in a highly loaded cluster (the lower the better). The scheduling delay is omitted to show how close the schedulers perform to the MILP solution regarding true cost calculation. 125
5.4 Comparison of deadline met percentage between the scheduling algorithms in both high load and light load period of the cluster (the higher the better). (a) and (b) shows the result when the deadline constraint is not considered. (c) and (d) shows the result when the deadline constraint is considered, and the jobs which are predicted to fail are removed from the job queue. 126
5.5 Cost comparison in large-scale simulation (the lower the better). FIFO incurs a very high cost as round-robin placements of executors lead to many active VMs simultaneously. 130
5.6 System Architecture. The resource managers communicate with the Mesos cluster manager through the REST APIs. The Mesos master is deployed in the local part of the cluster (local VM-1). 132
5.7 Cost Comparison between different scheduling algorithms (the lower the better). (a) shows the total cost incurred over a scheduling period, (b) shows the cumulative cost incurred over time. 135
5.8 Comparison of deadline met percentage between the scheduling algorithms (the higher the better). 136
5.9 Comparison of average job duration between the scheduling algorithms for different types of jobs (the lower the better). 137

6.1 The proposed RL model for the job scheduling problem, where a scheduling agent is interacting with the cluster environment. 152
6.2 Example scenarios for state transitions in the proposed environment. . . . 154
6.3 The workflow of the proposed environment in response to different agent actions. The red and green circles indicate the events which trigger negative and positive rewards, respectively, from the environment. 163
6.4 Convergence of the DQN algorithm. 168
6.5 Convergence of the REINFORCE algorithm. 169
6.6 Comparison of the total VM usage cost incurred by different scheduling algorithms in a scheduling episode. 171
6.7 Comparison between the scheduling algorithms regarding the average job duration in a scheduling episode. 173

6.8 Comparison of good placement decisions made by each scheduling algorithm in a scheduling episode. 174
6.9 The effects of the β parameter while using a multi-objective episodic reward in the RL environment. β=0.0 means time optimized only. β=1.0 means cost optimized only. Rest of the values represent a mix mode where both rewards have shared priority. 176

List of Tables

2.1 Comparison between the existing job scheduling algorithms for big data . 36

3.1 Related Work 45
3.2 Definition of Symbols 46

4.1 Related Work 70
4.2 Definition of Symbols 77
4.3 Experimental Cluster Details 88
4.4 Comparison of Average Scheduling Delays (unit: seconds) of different scheduling algorithms 96

5.1 Definition of Symbols 109
5.2 Simulation Cluster Details 121
5.3 VM Instance Pricing Models 121
5.4 Average Scheduling Delay (small-scale) 129
5.5 Average Scheduling Delay (large-scale) 131

6.1 Definition of Symbols 149
6.2 The action-event-reward mapping of the proposed RL environment. . . . 164
6.3 Cluster Resource Details 165
6.4 Hyper-parameters for DRL-agents and the environment parameters. . . . 165

Chapter 1
Introduction

BIG Data [1, 2] has recently gained enormous popularity due to its potential applications in science, business, government, and health domains. Scientific applications generate a large amount of data which is used for discoveries and explorations. Besides, social media data analysis, sentiment analysis and business data analysis are crucial for the business organizations to adapt customer needs and gain more profits. Therefore big data and analytics is becoming a common practice in organizations to get a competitive advantage. However, big data is not only about a massive amount of data, but it also covers the speed of incoming data and a variety of structured or unstructured data. Hence, even though handling big data analytics is necessary for both research and industry, putting them into practice is challenging.

Many prominent big data processing frameworks have emerged over the last decade. Google has introduced the MapReduce [3] programming paradigm in 2004. The most popular implementation of MapReduce was Apache Hadoop, and it was used almost everywhere to process batch-based large-scale big data applications. However, for streaming applications, frameworks like Apache s4 [4], Apache Storm [5] became popular. Recently, some hybrid big data processing frameworks were invented like Apache Spark [6] and Apache Flink [7], which support processing of both batch and streaming applications. To deploy a big data processing cluster, a huge amount of computing and storage resources are needed. However, buying and maintaining a massive amount of local resources is costly and difficult, especially for start-ups and small business organizations.

Cloud computing is an emerging platform which can provide infrastructure, platform and software for storing and computing of data. Nowadays Cloud computing is used in many small and large organizations like a utility [8] as it is more affordable to

go for the pay per use service of Cloud service providers rather buying and maintaining own computing resources. Therefore, Cloud computing is a viable option to deploy big data processing clusters [9]. With this approach, a set of Virtual Machines (VMs) can be hired from a Cloud service provider (CSP), then used to install a big data processing framework of choice. Thus, the Cloud resources are now utilized to set up a cluster which has a particular big data processing framework installed. Now, this cluster can be used to run a one or more of analytics applications in parallel. Thus, the VMs work as the worker nodes for the big data processing framework in the actual cluster.

Resource management is a broad term, which includes a lot of important elements of maintaining a big data processing cluster for running the applications effectively. The first key element is resource allocation, which means selecting and assigning the right amount of resources (e.g., VMs, CPU cores, and Memory) to a particular application. It is vital to allocate resources appropriately. Otherwise, the application might face severe performance degradation. The next and most crucial component is the scheduling of jobs. In a big data processing cluster, different users can have various applications which need to be executed. For example, a user can have a *Sort* application which sorts the words in a bunch of files. Each time this application is submitted, it is considered as a new job in the cluster. Therefore, the same application can be submitted multiple times as jobs, where each job can have varying resource demands or input data to process. When one or more jobs are submitted to the cluster for processing, depending on the resource allocation choices made before, the jobs need to be placed to actual machines. Here, depending on the job order, job priority, and VM placement, the performance and monetary cost of the cluster may vary. Other key elements of resource management are resource monitoring and logging, to check whether the health and the performance of the resources (VMs in this case for a Cloud-deployed) are degrading. Lastly, detecting, preventing, and recovering from anomaly and failures is also an aspect of resource management.

Even though the Cloud offers cheap, affordable and flexible resources, managing Cloud resources for big data applications is difficult. Many challenges need to be addressed when an organization shifts big data processing on the Cloud. These challenges include: deciding how much resources is needed for a job (a particular big data analytics

application), how to maximize the utilization of these resources to improve applications' performance, and how to reduce the monetary cost of resource usages. To meet these challenges, organizations often require experts who have good practical experience of managing Cloud resources as well as a proper understanding of that organization's application, data and various goals and objectives. In this thesis, we focus on a user-centric view, where the user can be an individual or a small/medium business organization to deploy a big data cluster on the Cloud. Specially, we explore how resource management techniques can be tailored to various user-demands such as performance improvement of the jobs, and deadline guarantee for the jobs; all while reducing the monetary cost of the whole cluster.

1.1 Challenges in Cost-efficient Resource Management

Reducing the monetary cost of a Cloud-deployed big data processing cluster, while maintaining a stable performance of the applications/jobs have brought many challenges. Here we discuss the key challenges addressed in the area of resource allocation and scheduling.

1.1.1 Challenges in Resource Allocation

Allocating appropriate amount of resource for an application is crucial as it determines whether a job has stable performance. In general, this is a manual task in a big data processing framework, so the users have to select the number of resources to allocate for each particular job. This often leads to over or under-provision of resources. In the first case, if the cluster is deployed in a public Cloud, over-provisioning can lead to an increase in monetary cost. In addition, for a cluster with a fixed set of resources, many jobs may starve. In the second case, under-provisioning can lead to severe performance degradation, along with the increase in deadline violations. Therefore, there is a need to develop a resource allocation mechanism which has the capability to select the right amount of resources to satisfy performance goals and deadline constraints, while reducing the monetary cost of the big data processing cluster. In addition, the resource

allocation mechanism needs to perform fine-grained resource allocation to maximize the potential of cost optimization. For example, a resource allocation mechanism which can allocate a varying chunk of resources (setting a different number of CPU cores and memory to be used for a task), can reduce the cost further than the ones which only choose a particular Virtual Machine (VM) to assign a task to.

1.1.2 Challenges in Dynamic Job Scheduling

Job scheduling is one of the most crucial components where a scheduler has to find the placement for one or tasks/executors of all the jobs. In addition, the capability to prioritize jobs based on their sensitivity (time-critical or not) is also crucial. Furthermore, if cluster resource availability changes, the scheduler needs to dynamically work and update with whatever resources available to schedule the upcoming jobs. From the users' perspective, the goal is to have a satisfying performance while reducing the monetary cost of the cluster. For example, tasks are generally placed in different Virtual Machines (VM). Therefore, if the scheduler can utilize the pricing model of the Cloud service provider, it can select cheaper VMs that provides an acceptable level of performance. Therefore, there is a need to design scheduling algorithms which are adaptive to cluster changes, can tackle the deadline of the time-critical jobs, and guarantee the required resources for the jobs.

1.1.3 Challenges in Job Scheduling in Hybrid Cloud

Due to the popularity of the Cloud services, many organizations are moving towards a Cloud deployment for their entire big data computing cluster. Although Cloud service providers offer flexible and affordable computing resources, for small or medium business organizations, moving everything towards the Cloud may not be ideal in terms of cost reduction. On the contrary, having a cluster deployed only with the on-premise resources would be difficult to scale, and also to adapt with peak demand surges. Therefore, a hybrid deployment can be an alternative solution, where the local resources of the cluster are used if the cluster is not overloaded and satisfactory performance can be maintained. Otherwise, the Cloud-deployed part of the cluster can be used to schedule

jobs. Therefore, there is a need to develop efficient scheduling algorithms which can leverage various pricing models of Cloud VMs, while considering the resource capacity of the local resources when scheduling jobs.

1.1.4 Challenges in Intelligent Learning of Job Scheduling Algorithms

To learn the workload and cluster characteristics, extensive job profiling information generated from the real cluster plays a key role to build the performance model. Then the performance models can be used in scheduling to match the workload type with the allocated resources. However, job profiling is an added overhead, and if the cluster/job characteristics change, the past performance models built from it will be obsolete. Thus, a scheduler designed for a particular environment cannot be used as a general solution to a wide variety of clusters due to the changing conditions of different resource availability, job demand and characteristics. As an example, a trivial heuristic or even an optimal job scheduling algorithm can not decide the type of placement which will be best suited for each particular job types. For example, a network-bound job may have performance improvement if the tasks are consolidated in fewer nodes to reduce inter-node communication delay. Therefore, a scheduler has to experience these issues and learn from it, and the learning process should be autonomic.

1.2 Research Problems and Objectives

This thesis focuses on cost-efficient resource management in Cloud-deployed big data processing clusters to maintain satisfactory performance. In order to address the challenges above, this thesis has identified and investigated the following research problems:

- **How to allocate resources in a fine-grained manner for a job, which incurs a lower monetary cost with a satisfactory level of performance?** The most common approach used is the static resource allocation, where the users have to set the resource requirements of each job manually. However, it leads to over or under-provisioning of resources. There has been some attempt to build performance models from job profiles. However, the existing techniques lack in fine-

grained resource allocation for tasks in Cloud VMs. Thus, a resource allocation mechanism is needed, which can utilize the Cloud VM pricing models, and allocate fine-grained resources for each job in such a way that the job does not violate the deadline constraint.

- **How to schedule jobs dynamically in a Cloud-deployed cluster, while reducing the overall Virtual Machine (VM) usage cost of the cluster?** The default framework schedulers and the other existing works prioritized the performance of the jobs, but not the monetary cost reduction of the whole cluster if Clouds VMs are used as worker nodes. Besides, job arrival is often stochastic in nature, and a scheduler has to make a fast decision. Thus, there is a need to design fast scheduling algorithms which can adapt to dynamic cluster resource changes, utilize the VM pricing models while placing tasks, all while maintaining a satisfactory performance for each job.

- **How to efficiently schedule jobs in a big data cluster deployed in the hybrid Cloud while reducing the monetary cost?** The framework scheduler assumes that the deployment of a cluster is with homogeneous worker nodes (all the machines have the same resource capacity). However, in a hybrid deployment, VMs can be of different sizes along with various pricing models. Also, the cost of using a local VM and a Cloud VM can be different even if they have similar performance. Therefore, there is a need to devise scheduling algorithms which can leverage from the pricing models of the Cloud service providers and utilize different VM instance sizes to reduce the monetary cost of the hybrid Cloud-deployed cluster.

- **How to design an intelligent scheduling algorithm which can automatically learn the inherent job and cluster characteristics, while optimizing one or more objectives?** The heuristics and optimization model-based algorithms can only address a specific target objective and cannot capture the inherent cluster and job characteristics. Also, they require fine-tuning and human intervention depending on different goals, scenario, and setup. Therefore, there is a need to design intelligent scheduling algorithms, which can incorporate the power of modern AI-based learning (specifically, Reinforce Learning or RL), to learn and utilize various inher-

ent cluster characteristics while optimizing one or more target goals. Besides, there is a need to reduce human intervention, and also a need for algorithms which can be generalized to work in a wide range of scenarios.

1.3 Evaluation Methodology

To evaluate the effectiveness of the proposed approaches in the thesis, we have used the Nectar[1] Research Cloud to deploy our experimental big data processing cluster. Besides, we have used real-world workload traces to extract job arrival times of the clusters to incorporate various job scheduling scenario. The jobs used in different experiments are real-world benchmark applications such as WordCount, Sort, and PageRank; where all these applications have different performance requirements and show various performance variability in the system.

For the efficacy of the experimental evaluation and comparison with the baseline algorithms, we have developed different prototype systems for every piece of work. Also, to test the scalability of the proposed algorithms, we have implemented simulation tools that can be used to evaluate various scheduling policies. For the sake of repeatability of the experiments, and also for the use of the research community, all of the implementations of the algorithms, tools, simulators, and framework are made open source[2].

Recently, in-memory big data processing frameworks became popular due to their speed of data processing capabilities. In particular, Apache Spark is the most popular framework as it is more suited for Machine Learning (ML) and iteration-based analytics jobs. Thus, our approaches focus on solving the resource management problems from the perspective of Apache Spark's architecture, and it's deployment on the public Cloud resources.

1.4 Thesis Contributions

The **key contributions** of this thesis are listed below:

[1] www.nectar.org.au
[2] https://github.com/tawfiqul-islam

1. A survey and taxonomy on resource management of big data applications in the Cloud environment, with a specific focus on resource allocation and job scheduling algorithms.

2. A deadline-based cost-optimized Cloud resource allocation mechanism for big data applications.

    - A mathematical model that predicts the completion time of a big data application based on the number of executors and other properties.
    - A resource allocation model where a cost-effective, deadline-based Resource Allocation Scheme (RAS) can be found for an application.
    - A job profiler to profile any application concerning varying input workloads, iterations, and resource allocation schemes.

3. Dynamic job scheduling algorithms for Cloud-deployed big data processing clusters.

    - Two job scheduling algorithms. The first algorithm is a greedy algorithm adapted from the Best-Fit-Decreasing (BFD) heuristic, and the second algorithm is based on Integer Linear Programming (ILP). Both of these algorithms can improve the cost-efficiency of a Cloud-deployed cluster. Besides, the algorithms also prioritize jobs based on their deadlines and enhance job performance for network-bound jobs.
    - A scheduling framework by utilizing Apache Mesos [10] cluster manager and this framework can be used to implement scheduling policies for any Mesos supported data processing frameworks.

4. SLA-based scheduling algorithms for big data computing clusters deployed on hybrid Cloud.

    - A mathematical model of SLA-based scheduling in a hybrid Cloud.
    - Two job scheduling algorithms. The first algorithm is a modified version of the First-Fit (FF) heuristic for solving bin packing problems. The second algorithm uses a greedy approach to iteratively find the cost-optimal placement

for each executor of a job. Both algorithms can improve the cost-efficiency of a hybrid cluster.

- An event-based simulator that can be used to simulate, test, and compare different job scheduling policies.

5. Intelligent scheduling algorithms that can automatically learn the inherent cluster and job characteristics, and optimize one or more target objectives.

    - An RL model of the job scheduling problem of big data applications in Cloud computing environments. The reward formulation is also provided for the training of DRL-based agents to satisfy resource constraints, optimize cost-efficiency, and reduce average job duration of a cluster.
    - A prototype of the RL model in a python environment which is plugged into the TF-Agents[3] framework.
    - Two DeepRL-based agents, DQN and REINFORCE, which are trained as scheduling agents in the TF-agents framework.

1.5 Thesis Organization

The structure of this thesis is shown in Figure 1.1. The remaining part of this thesis is organized as follows:

- Chapter 2 presents a taxonomy and literature review on resource management for big data applications in Cloud environments. It mainly focuses on various resource allocation and scheduling approaches. This chapter is derived from:
  - **Muhammed Tawfiqul Islam** and Rajkumar Buyya, ”Resource Management and Scheduling for Big Data Applications in Cloud Computing Environments”, *Handbook of Research on Cloud Computing and Big Data Applications in IoT*, B. Gupta and D. Agrawal (eds), 1-23pp, ISBN-13: 978-1522584070, IGI Global, USA, 2019.

- Chapter 3 presents a cost-efficient, fine-grained, and deadline-aware resource allocation mechanism. This chapter is derived from:

[3] https://www.tensorflow.org/agents

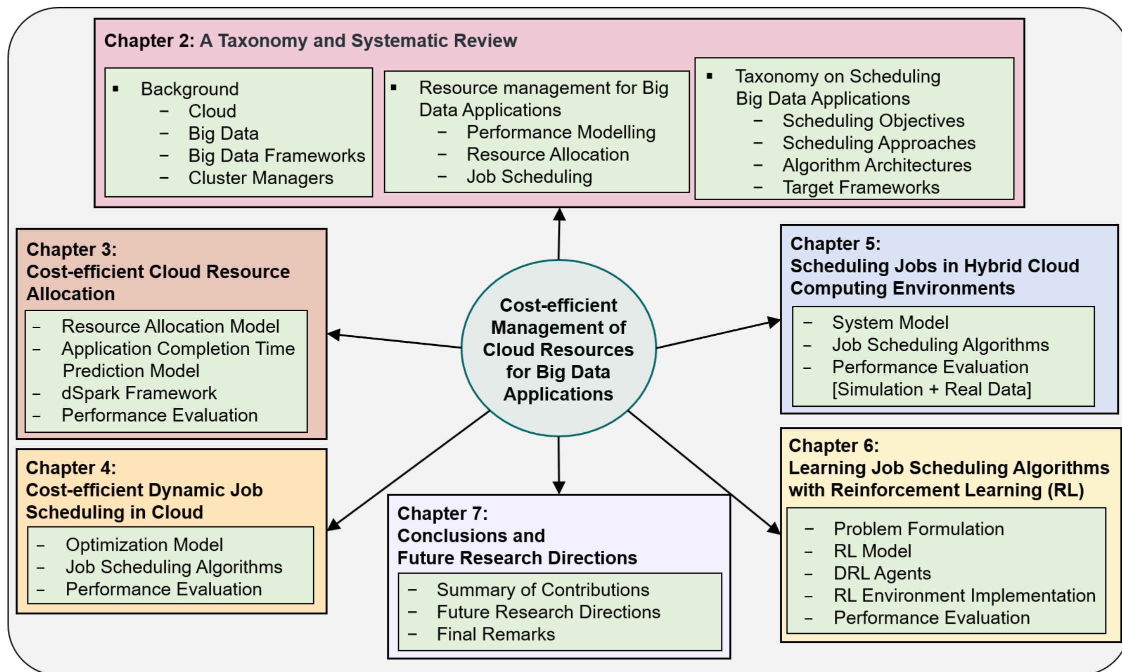


Figure 1.1: Thesis Organization

- **Muhammed Tawfiqul Islam**, Shanika Karunasekera, and Rajkumar Buyya, ”dSpark: Deadline-based Resource Allocation for Big Data Applications in Apache Spark”, Proceedings of the 13th IEEE International Conference on e-Science (e-Science 2017), Auckland, New Zealand, October 24-27, 2017.

- Chapter 4 presents dynamic cost-efficient scheduling algorithms to schedule jobs in a Cloud-deployed big data processing cluster. This chapter is derived from:
  - **Muhammed Tawfiqul Islam**, Satish N. Srirama, Shanika Karunasekera, and Rajkumar Buyya, ”Cost-efficient Dynamic Scheduling of Big Data Applications in Apache Spark on Cloud”, *Journal of Systems and Software (JSS)*, Volume 162, Pages: 1-14, ISSN: 0164-1212, Elsevier Press, Amsterdam, The Netherlands, April 2020.

- Chapter 5 presents job scheduling algorithms to be used in a cluster deployed in a hybrid Cloud, which leverage the pricing model and utilize the local and Cloud VMs of a cluster effectively to reduce monetary cost. This chapter is derived from:
  - **Muhammed Tawfiqul Islam**, Huaming Wu, Shanika Karunasekera, and Rajkumar Buyya, ”SLA-based Scheduling of Spark Jobs in Hybrid Cloud Computing

Environments", *IEEE Transactions on Computers (TC)* [Under 2nd Revision].

- Chapter 6 presents efficient Deep Reinforcement Learning (DRL) based scheduling agents which can automatically learn various inherent cluster and job characteristics to optimize one or multiple objectives. This chapter is derived from:
  - **Muhammed Tawfiqul Islam**, Shanika Karunasekera, and Rajkumar Buyya, "Cost and Performance-oriented Spark Job Scheduling in Cloud with Deep Reinforcement Learning", *IEEE Transactions on Parallel and Distributed Systems (TPDS)* [Under Review].
- Chapter 7 concludes the thesis, summarizes the key contributions and highlights future research directions.

Chapter 2

A Taxonomy and Systematic Review

This chapter presents software architectures of the big data processing frameworks. It also provides in-depth knowledge on resource management techniques involved while deploying big data processing frameworks in the Cloud environment. It starts from the very basics and gradually introduce the core components of resource management which we have divided into multiple layers. It covers the state-of-art practices and researches done in SLA-based resource management with a specific focus on the job scheduling mechanisms.

2.1 Introduction

CLOUD Computing is an emerging platform which can provide infrastructure, platform, and software for storing and computing of data. Nowadays Cloud Computing is used in many small and large organizations like a utility [8] as it is more affordable to go for the pay per use service of Cloud service providers instead buying and maintaining own computing resources. While registering in any Cloud Service, both the Cloud service customer and the Cloud service provider must agree on some predefined policies which are called the Service Level Agreement (SLA). Violation of SLAs may affect the proper execution and performance of an application of any customer, so it poses a significant threat on a Cloud service provider's business reputation. Therefore, it is essential to manage the Cloud resources in such a way that it guarantees SLA.

This chapter is derived from:

- **Muhammed Tawfiqul Islam** and Rajkumar Buyya, "Resource Management and Scheduling for Big Data Applications in Cloud Computing Environments", *Handbook of Research on Cloud Computing and Big Data Applications in IoT*, B. Gupta and D. Agrawal (eds), 1-23pp, ISBN-13: 978-1522584070, IGI Global, USA, 2019.

Big Data [1, 2] is the recent hype in information technology. Scientific applications generate a large amount of data which is used for discoveries and explorations. Besides, social media data analysis, sentiment analysis, and business data analysis are crucial for business organizations to adopt customer needs and gain more profits. Cloud computing can be an appropriate solution to host big data applications, but many challenges need to be addressed to use the existing Cloud architectures for big data applications. This chapter discusses the challenges of hosting big data processing framework in the Cloud. Moreover, it also gives a comprehensive overview of Cloud resource management for big data applications. Resource management is a broad domain that contains many complex components. However, to make it easier to understand, we divide it as a layered architecture and discuss the critical elements from each layer. Our focus will be on resource allocation and scheduling mechanisms and how the existing research tried to incorporate SLA in these components. We will also point out the limitations of the current approaches and highlight future research directions.

In summary, the **contributions** of this chapter are as follows:

- Basic understanding of Cloud computing and big data processing framework.
- Knowledge about the software architectures and use-cases of different big data processing frameworks.
- Knowing about the vital software and tools for successful deployment and management of big data clusters .
- Learning the key steps and significant components of resource management for big data applications through a layered architecture.
- Focusing on the existing literature on SLA-based application/job scheduling mechanisms and finding the applicability of these techniques in different application scenarios.
- Gaining insights about the research gaps and highlights on the future research challenges.

The contents of this chapter are organized as follows. Section 2.2 provides background on Cloud computing, big data, big data processing frameworks and their archi-

tectures and some popular cluster managers. Section 2.3 gives a layered overview of the overall resource management process for big data applications on the Cloud. Section 2.4 presents a taxonomy of job scheduling mechanisms for big data applications. Finally, section 2.5 concludes the chapter.

2.2 Background

In this section, we briefly discuss the key features of Cloud computing. Moreover, we explain the architectures of the popular open-source software systems for processing big data applications. Also, we provide an overview of some popular cluster managers. Finally, we conclude with explaining why the Cloud is a viable alternative to deploy a big data processing software and how cluster managers can be used for efficient management of the system.

2.2.1 Cloud Computing

Cloud computing delivers a shared pool of on-demand computing resources on a pay-per-use basis. The main features of Cloud computing are:

1. Resource elasticity: Cloud resources can be easily scaled up or down to meet application or user demands.

2. Metered service: users are billed based on what resources they used and how long they have used them.

3. Easy access: the resources of Cloud can be easily accessed and can be provisioned as a self-service manner.

There are three different types of Cloud. These are:

1. Public Cloud: There are many public Cloud service providers who offers computing resources as a pay-per-use basis. Organizations can hire resources from these service providers to deploy their own applications. It greatly reduces the cost of buying computing hardware and removes the burdens of managing local resources.

2. Private Cloud: Many organizations setup an on-premise computing resource facility which is known as the private Cloud. The main reason for setting up a private Cloud is to reduce the data transfer overhead to the public Cloud. In addition, it also ensures that private and sensitive data are kept on the organization's premises to reduce security threats.

3. Hybrid Cloud: It is mix of both public and private Cloud. Sometimes organizations need to scale up their resources in public Cloud to be processed in the public Cloud.

Cloud provides computing as a service, and we can divide Cloud services in three ways:

1. Software as a Service (SaaS): SaaS can be used from any devices through the Internet and typically these services are accessed via a web browser. The required software needed by the user for any specialized task are already developed and provided thorough different interfaces. Users just define the task, input data and collect the results. Example: Google Apps

2. Platform as a Service (PaaS): A platform is provided for developing distributed, scalable Cloud-based programs. It greatly reduces the hassle for managing the underlying resources. Example: IBM Cloud

3. Infrastructure as a Service (IaaS): Computing and storage resources are provided to setup a user's own infrastructure to build platform and services. Reduces the hassle of buying and managing own physical hardware, provides a scalable on demand pool of resources. Example: Windows Azure, Amazon EC2.

2.2.2 Big Data

In today's world, huge amount of data is being generated through social media, scientific explorations and many other emerging applications like Internet of Things (IoT).

---

https://gsuite.google.com.au/intl/en_au/
https://www.ibm.com/cloud/
https://azure.microsoft.com/en-au/
https://aws.amazon.com/ec2/

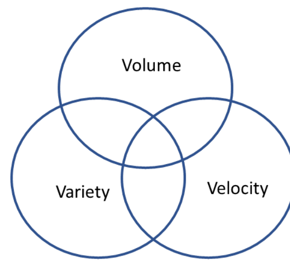


Figure 2.1: Big Data 3V

The term "Big data" is not about the size of data; rather it covers many other aspects. For simplicity, we can define it in terms of the 3V as shown in Figure 2.1.

The volume of data can be small or large, from a few Megabytes to thousands of Terabytes. Each day we are generating so much data that recently (in 2015) we have moved into a Zettabyte era. Velocity represents the speed of incoming data. For example, some applications need real-time or near real-time processing and comes with great speed. These types of applications can be categorized as streaming applications. In contrast, applications that need offline processing of huge volume of static data are called batch applications. Finally, data can have many varieties such as structured, unstructured etc. Storing and processing of data is often not possible by the traditional Database Management Systems (DBMS) and NoSQL has greatly replaced SQL in many domains. There are many other aspects of big data (many other Vs) depending on the specific domain.

2.2.3 Big Data Processing Frameworks

Processing big data is a difficult task, and it is not possible in a centralized system. Therefore, distributed computing solutions are used for parallel processing of big data. Many big data processing frameworks have emerged over the last decade. Figure 2 shows a taxonomy on big data processing frameworks. As it shows, previously only batch-based frameworks like Hadoop was mostly used. However, due to the discovery of many scientific, business and social streaming applications, real-time processing became more

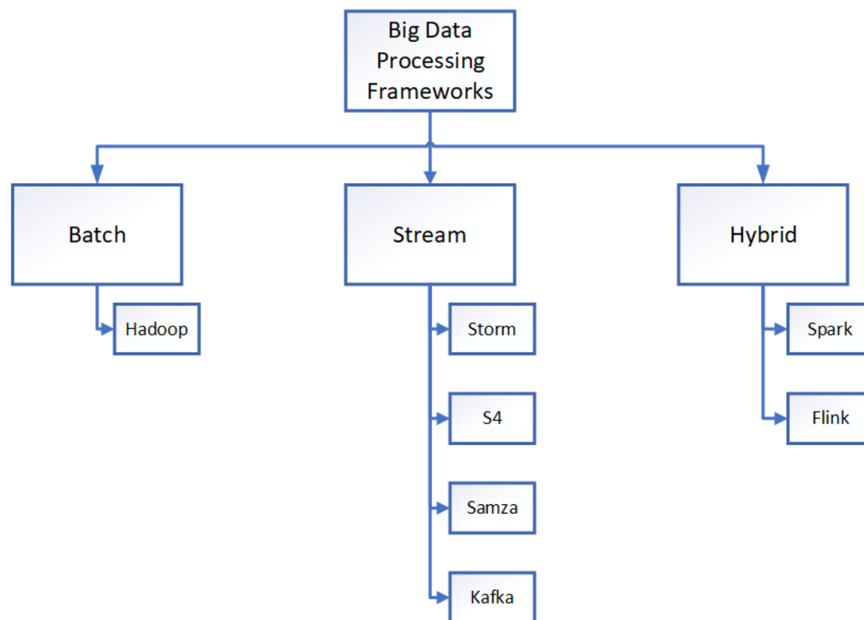


Figure 2.2: A Taxonomy of Big Data Processing Frameworks

influential and dedicated stream processing frameworks like Strom, S4 were invented. However, applications became more complex, and often organizations need to have both batch and stream-based processing. Hence, some hybrid processing frameworks like Apache Spark, Apache Flink are being used in the industry.

In this chapter, we only focus on batch and hybrid-based processing frameworks and briefly discuss about some of the most popular ones.

Apache Hadoop

Apache Hadoop, introduced by Yahoo in 2005, is the open source implementation of the MapReduce programming paradigm. The main feature of Hadoop is to use primarily distributed commodity hardware to parallel processing of batch-based jobs. The core of Hadoop is its fault-tolerant file system Hadoop Distributed File Systems (HDFS) [11] that can be explicitly defined to span in many computers. In HDFS, the block of data is much larger than a traditional file system (4KB versus 128MB). Therefore, it reduces the memory needed to store the metadata on data block locations. Besides, it reduces the seek operation in big files. Furthermore, it greatly enhances the use of the network as

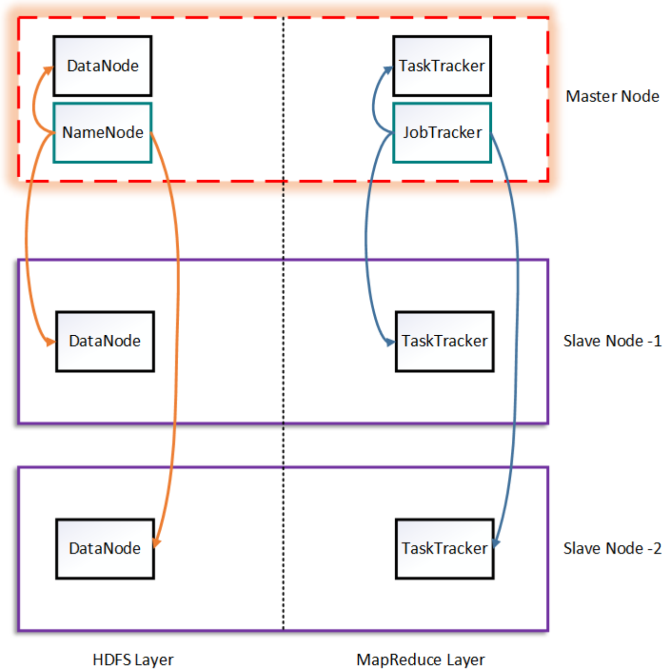


Figure 2.3: Apache Hadoop Architecture

only a fewer number of network connections are needed for shuffle operations. In the architecture of HDFS, there are mainly two types of nodes: Name node and Data node. Name node contains the metadata of the HDFS blocks, and the data node is the location where the actual data is stored. By default, three copies of the same block are stored over the data nodes to make the system fault tolerant. The resource manager of Hadoop is called Yarn [12]. It is composed of a central Resource Manager who resides in the master node and many Node Managers that live on the slave nodes. When an application is submitted to the cluster, the Application Master negotiates resources with the Resource Manager and starts container (where actual processing is done) on the slave nodes.

The main drawback of Hadoop was that it stored intermediate results in the disk, so for shuffle-intensive operations like iterative machine learning, a tremendous amount of data is stored in the disk and transferred over the network which poses a significant overhead on the whole system.

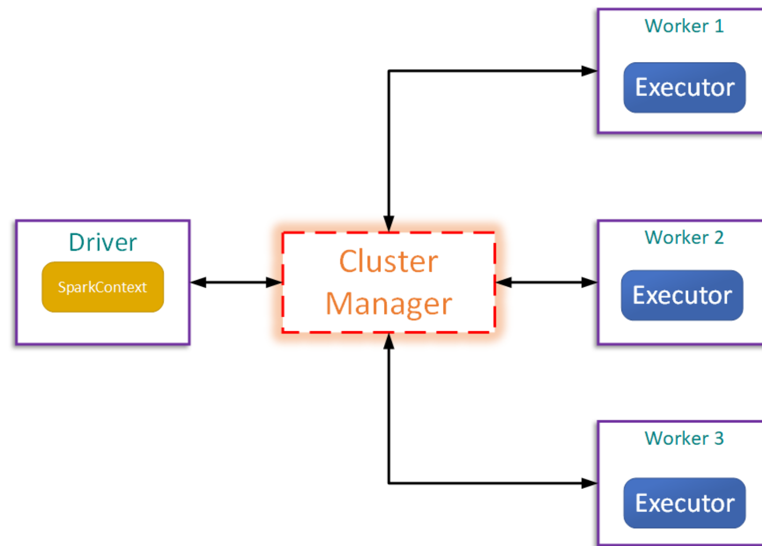


Figure 2.4: Apache Spark Architecture

Apache Spark

Apache Spark [13] is one of the most prominent big data processing frameworks. It is an open source, general-purpose, large-scale data processing framework. It mainly focuses on high-speed cluster computing and provides extensible and interactive analysis through high-level APIs. Spark supports batch or stream data analytics, machine learning and graph processing. It can also access diverse data sources like HDFS, HBase [14], Cassandra [15], etc. and use Resilient Distributed Dataset (RDD) [16] for data abstraction.

As compared to the Hadoop system tasks, Apache Spark allows most of the computations to be performed in memory and provides better performance for some applications such as iterative algorithms. When the results do not fit on the memory, the intermediate results are written to the disk. Spark can run locally in a single desktop, in a local cluster, and on the Cloud. It runs on top of Hadoop Yarn, Apache Mesos [10] and the default standalone cluster manager. Jobs/applications are divided into multiple sets of tasks called stages which are inter-dependent. All these stages make a directed acyclic graph (DAG), where each stage is executed one after another.

Apache Flink

Apache Flink [7] is an open-source stream processing framework. It executes data-flow programs in data-parallel pipelines. Flink is fault-tolerant and treats batch data as a form of a stream, therefore, it is a hybrid framework. Programs can be written in Java, Scala, Python, and SQL. Flink does not provide any data storage mechanism. Instead, it uses other data sources like HDFS, Cassandra, etc. During the execution stage, Flink programs are mapped to streaming dataflows. Every dataflow starts with one or more origins (input, queue or file system) and ends with one or more sinks (output, message queue, database or file system). An arbitrary number of transformations can be done on the stream. These dataflow streams are arranged as a directed acyclic dataflow graph, allowing the flexibility for the applications to branch and merge dataflows. Flink is relatively new and unstable as compared to the matured frameworks like Hadoop and Spark. It is yet to be seen whether Flink can be scalable like Spark in a production-grade cluster.

2.2.4 Cluster Managers

Apache Hadoop Yarn

Apache Hadoop Yarn [12] is the resource manager for Apache Hadoop. The core idea of Yarn is to split up the mechanisms for resource management such as job scheduling, monitoring, etc. into separate daemons. There is a global Resource-Manager in the master node and Node Managers in each of the worker/slave nodes. Resource Managers and Node Managers form the whole data-computation framework. Resource Manager is the ultimate co-ordinate that can dictate resource provisioning and scheduling in the entire system. Node Managers are responsible for running containers and monitor resource usages and reporting the resource usage statistics to the Resource Manager. Furthermore, per-application Application-Manager negotiates with the Resource Manager to reserve resources and collaborates with the Node Manager to run containers and monitor the tasks.

The Resource Manager has two main components: Scheduler and Applications-

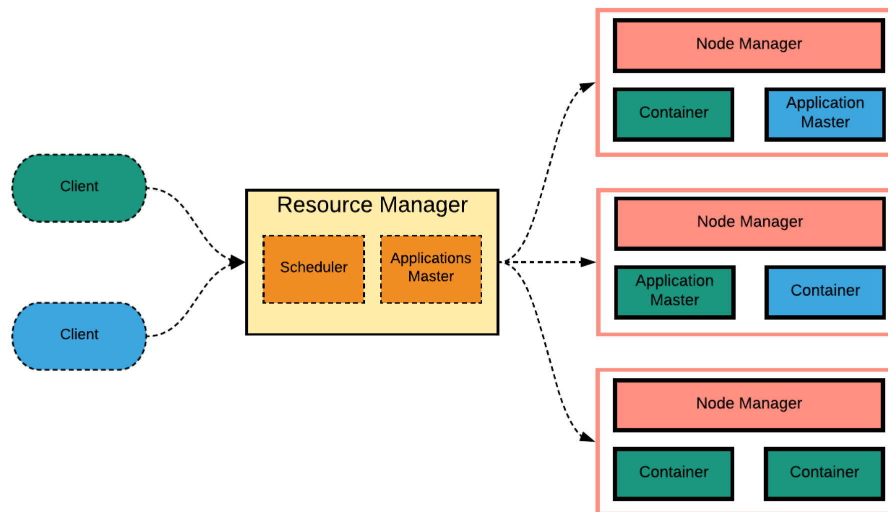


Figure 2.5: Apache Hadoop Yarn Architecture

Manager. Scheduler tracks and maintains a queue of jobs set the order of the jobs and allocate resources to each of the jobs before execution. The scheduler functions are based on the implemented policies and SLA requirements of the applications. The scheduler has a pluggable policy which makes it extendable to different scheduling policies. For example, CapacityScheduler and FairScheduler are the example plugins implemented and available with Yarn. Applications-Manager accepts job submission requests and provides the service to restart failed jobs.

Apache Mesos

Apache Mesos [10] is said to be the data-center level cluster manager. Mesos was built primarily to support multiple different big data processing frameworks to be running in the same cluster. Mesos isolates the resources (e.g., CPU, Memory and disk) shared by different framework tasks/executors and run them in the same physical/virtual machine. Schedulers from different frameworks negotiate with Mesos to reserve resources for running tasks. Moreover, each application (of any big data processing system like Spark, Hadoop, Storm) is called a framework and can have a custom implemented scheduler that can negotiate with Mesos to set the required resources for that appli-

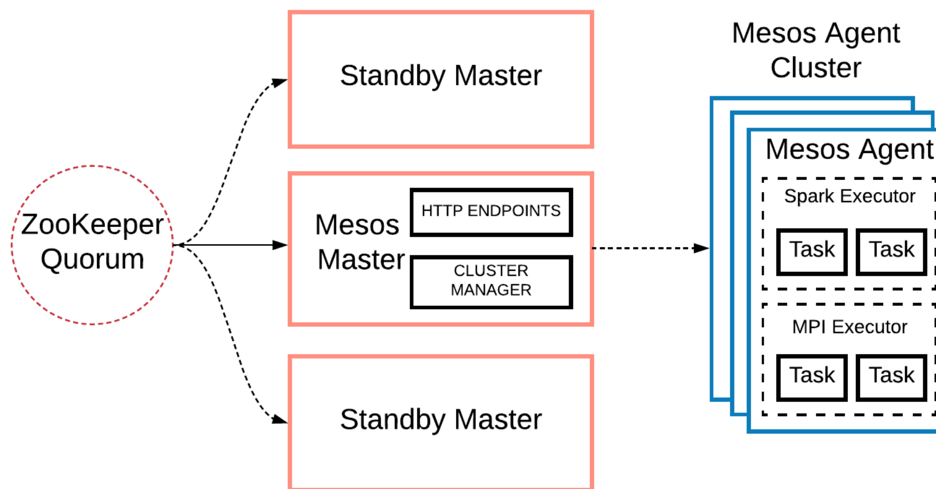


Figure 2.6: Apache Mesos Architecture

cation.

Mesos send resource offers to each framework by using the Dominant Resource Fairness (DRF) [17] resource allocator which tries to distribute the resources among multiple frameworks equitably. However, Mesos has an advanced scheduler and operator HTTP APIs and supports Dynamic Resource Reservation for any application. Therefore, by using the scheduler/operator APIs, it is possible to build custom pluggable scheduler with specific SLA requirements. Frameworks can also be assigned with particular roles and set resource quotas to make the resource management flexible.

Google Kubernetes

Kubernetes is an open source container management platform which is designed to run at production scale. It was built upon the foundations laid by Google. The architecture of Kubernetes supports loosely-couped mechanism for service discovery. There are a master and one or more computing nodes in a Kubernetes cluster. The master exposes APIs, schedules workloads and controls the cluster. Each node runs a container runtime like Docker or rkt agent that communicates with the master. A node also has additional

https://kubernetes.io/

components responsible for logging, monitoring, service discovery and optional add-ons. A pod is a collection of containers that serve as a core unit of management. It acts as logical isolation for containers sharing same context and resources. Replica sets provide the required scale and availability of services by maintaining a pre-defined set of pods. The deployment of an application can be scaled by using replica sets which ensures an application has its desired number of pods running to meet the requirements. The master node has etcd, which is an open-source distributed key-value database and acts as the single point of truth for all components in a Kubernetes cluster. When an application gets enough pods to run, the nodes pull images from the image registry and works with the local container runtime to launch the container in each pod. Kubernetes is flexible and provides a rich set of APIs for building custom container management modules which are particularly useful in deploying efficient, large-scale IoT/Fog based applications.

2.3 Resource Management for Big Data Applications

In this section, we will provide a brief overview of the significant components of resource management for Big Data applications. Many steps or components can be included. However, the overall process of managing resources for big data applications is a complex task, and many parts are inter-dependent thus it is hard to distinguish them. Therefore, as shown in Figure 2.7, we have simplified the categorization in three different layers and only discuss the key elements from each of these categories.

2.3.1 Setup Layer

The first layer of resource management is the Cluster Setup. In this layer, hardware or virtualized resources are selected depending on the applications. Additionally, a cluster manager is deployed to manage the resources and jobs from different big data processing frameworks. Lastly, one or more big data processing frameworks are used.

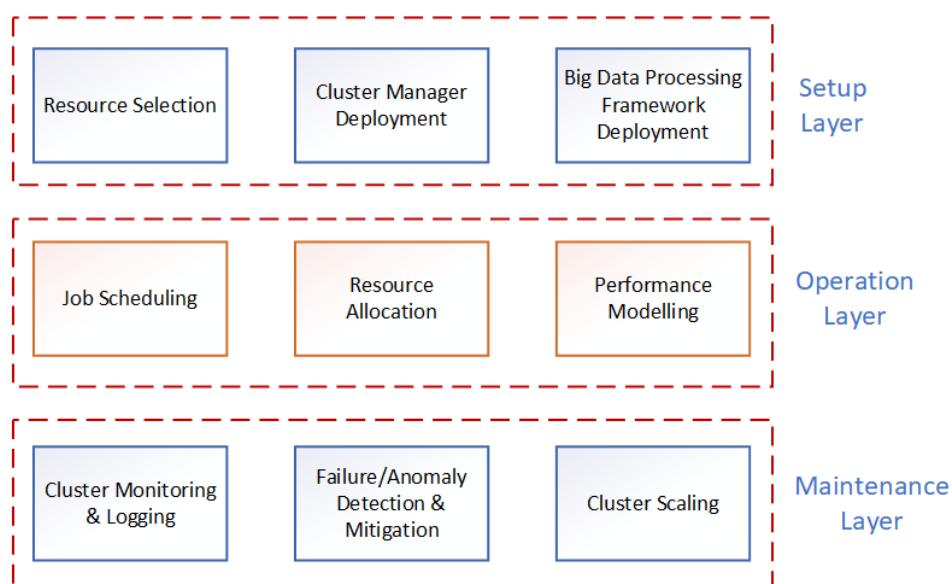


Figure 2.7: Key components of Resource Management in a Big Data Cluster

Resource Selection

Both physical or virtualized resources can be used to build a cluster. Generally, depending on the applications and analytics demands of any business organization, the hardware resources are chosen. The setup can be done on-premise (local cluster or private Cloud), deployed on Cloud resources (public Cloud) or a hybrid deployment (some local resources with a pay-as-you-go subscription from a Cloud provider) can also be made. The actual underlying hardware resources might vary with applications. However, CPU, RAM, Storage, and Network are the must no matter where the cluster is deployed. Nowadays, GPU resources are gaining popularity due to the widespread use in sophisticated machine learning (deep learning) algorithms running in platforms like TensorFlow.

Cluster Manager Deployment

The next step is to choose a cluster manager to manage both the jobs and the resources. A cluster manager also balances the workloads and resource shares in a multi-tenant envi-

https://www.tensorflow.org/

ronment. For containerized applications, Kubernetes or Docker Swarm can be deployed to provide container management platform. Kubernetes excels as a complete management system featuring scheduling, dynamic on-the-fly updates, auto-scaling, and health monitoring. However, Docker Swarm features a system-wide view of the whole cluster from a single Docker engine. Apache Hadoop Yarn is the cluster manager of choice if all the applications of the cluster are only MapReduce or Hadoop-based. In contrast, Apache Mesos is a better choice than Yarn as it supports efficient resource isolation for multiple different big data processing frameworks and provides strong scheduling capabilities.

Big Data Processing Framework Deployment

Many big data processing frameworks are available which can run distributed applications across one or more clusters. The applications can be real-time, stream or batch and for each type of applications, there are some frameworks which are capable of handling the requirements efficiently. It is not possible to say which is the best possible framework to deploy in general. Instead, each one has its own merits and suits a group of applications. For example, in the last decade, Hadoop was the most prominent framework to process MapReduce based static batch jobs. However, due to the increasing popularity of real-time systems and streaming applications; Apache Spark, Apache Flink, and Apache Storm have become the standard choice to tackle them. Apache Storm is particularly useful for stream-based applications. Apache Spark is vastly replacing both Hadoop and Storm, and it is a hybrid framework that supports both batch and stream processing. Apache Flink is new a hybrid framework that needs to be more stable to compete with the likes of Spark or Storm.

2.3.2 Operation Layer

The second layer of resource management is the operation layer. Here, performance models are built to determine the set of resources to be allocated that is enough to meet

---

https://docs.docker.com/engine/swarm/

user SLA and schedule multiple jobs in a multi-tenant setup. Moreover, the overall cluster utilization is maximized, and each jobs performance is enhanced without interfering with any other jobs SLA.

Performance Modelling

The performance of a job might vary depending on various aspects like allocated resources, workload size, task placement, etc. Hence, before a complete deployment of a job, performance models can be established which will be used in resource allocation and job scheduling phase to choose an optimized set of resources to run the job without sacrificing any performance constraints. Generally, performance modeling can be done in two ways. First, running the job with different resource configurations and workloads to build job profiles. Second, collecting historical data of jobs running in the cluster. Both job profiles or historical data can be used to perform statistical analyses, training machine learning algorithms or build mathematical models. These models are then used to select optimal resource configurations and efficient scheduling strategies. There are many existing researches that tried to model the performance of different types of jobs running in both Spark or Hadoop based frameworks. [18] modelled the performance of MapReduce workloads in a heterogeneous cluster (where resources are different types, or the performance varies). This model is then used to predict the job completion times. [19] proposed a simulation-based approach where they have used different Apache Spark configuration parameters and modeled different stages of a job to predict its completion time.

Resource Allocation

Resource allocation means reserving a set of resources for a job which will be used by that job to run its tasks up to a specific period. Generally, resource allocation is of two types.

1. Static: Manual resource allocation for each job by the user if the user has enough knowledge on the application behavior on the cluster environment.

2. Dynamic: The job is started with a few sets of resources. Based on the utilization and to meet the SLA constraints, more resources might be allocated or deallocated over time.

Choosing the right amount of resources to meet user SLA is crucial as improper resource allocation might lead to either under-utilization or over-utilization problem. Therefore, as mentioned in the previous step, performance models are used to determine the optimal set of resources for each job. Resource allocation can be done from both big data processing framework or cluster manager side. [20] modeled Spark jobs based on different parameters such as input size, iteration, resource requirements to predict the job runtime. Then an optimized resource configuration parameter is suggested based on the models which is enough for that job to meet its deadline. [21] also suggested a deadline-aware model to perform resource allocation which is also cost-effective. The model is called OptEx and it estimates job runtime before resource allocation by using the job profiles. [22] proposed a resource provisioning framework for MapReduce jobs which also uses job profiles from jobs to estimate the required resources for jobs.

Job Scheduling

It is the most critical component of resource management. Job scheduling means settings the order of the jobs in which they will run on the cluster. Additionally, the resources can also be ordered before running any jobs. Both job and resource ordering depend on the scheduling policy. The most straightforward scheduling policy that is used in all the cluster managers and big data frameworks is the FIFO (First in First Out). Here, jobs are ordered according to their arrival time; that means the job that comes first is executed first in the cluster. If there are not enough resources in the cluster to run all the jobs, then the remaining jobs are placed in a queue which is sorted based in increasing order of their arrival time. In most cases, the FIFO scheduler under-performs with complex SLA requirements in a multi-tenant cluster setup. Therefore, a vast amount of research exists in this area that proposes efficient schedulers with optimal scheduling policies. However, most of the scheduling algorithms are either application or the SLA-demand specific. Also, the parameters that are considered vary greatly depending on the applica-

tion or cluster setting. The more sophisticated schedulers tackle both resource allocation and scheduling together to make it more efficient. First, these schedulers use some pre-existed performance models for the jobs at hand or build it dynamically then decide the resource configuration for a job before scheduling it. Moreover, the resource usages of the currently executing jobs are tracked, and further resources will be reallocated or deallocated to make it optimally achieve the SLA requirements. Job scheduling is a massively broad and explored topic in both big data and Cloud computing. We will provide a detailed discussion and compare the existing works on job scheduling.

2.3.3 Maintenance Layer

It is the final layer of resource management for big data applications. The components of this layer are responsible for maintaining an already deployed big data cluster.

Cluster Monitoring and Logging

Cluster Monitoring is crucial as it plays a vital role in the resource management lifecycle. The cluster monitoring data can be logged and saved in persistent storage. This data can be used to validate the performance of the resource allocation and scheduling policies. Besides, if a feedback-based system is used (can be both feedback-based resource re-provisioning/ scheduling and machine learning models that are updated and improved by using the current system status), it needs to use the cluster monitoring data to improve the system performance. Popular big data processing frameworks like Hadoop, Spark, and Storm provide cluster-wide monitoring data and web-UI to visualize the health of the cluster. Besides, cluster monitoring data can also be found from cluster/container management systems like Kubernetes, Mesos and Yarn. Sometimes while building sophisticated application/user-specific resource management modules, data from the underlying framework may not be enough. In those cases, the administrator or developer might need fine-grained resource usage and health data which is possible by using tools such as Collectd or Prometheus . A cluster monitoring system

https://collectd.org/
https://prometheus.io/

like Prometheus not only provides cluster monitoring data, but it can also offer a time-series database to store the monitoring data. The database is particularly useful for applying advanced machine learning algorithms or performing time-series analysis on the monitoring data.

Failure/Anomaly Detection & Mitigation

When a cluster is deployed, and in operation, jobs might fail due to an anomaly in the system, hardware/software failure, resource over-utilization, resource-scarcity, etc. By analyzing cluster-wide monitored log data, it is possible to detect the root cause of failures in the system. It is important to solve the issue to keep the cluster healthy so that the jobs can meet their SLAs. The most trivial way to solve the failure is to restart the failed jobs. In the case of resource scarcity, jobs might fail due to a shortage of resource or interference of co-located jobs. This problem can be solved by throwing more resources in the cluster so that the jobs can run properly. In case of hardware or software failures, the affected hardware that might be prone to failure can be avoided in scheduling to avoid any further failures. Chronos [23] is a Hadoop-based failure-aware scheduler that uses pre-emption on failed jobs. Then it recovers from failure by reallocating the failed jobs with pre-empted resources to meet the SLA objectives. Fuxi [24] is a fault-tolerant resource management and scheduling system that can predict and prevent failures in large clusters to satisfy user performance needs.

Cluster Scaling

A big data processing cluster might need to be scaled up and down based on the current usage. In a high-load hour, the currently running VMs might not be enough to run all the jobs while satisfying all the users SLAs. Therefore, in this situation, the cluster needs to be scaled up to satisfy the peak surge of resource demands. In contrast, in a light-load hour, a cluster might go under-utilized. In this scenario, the existing cluster jobs can be consolidated in fewer VMs so that the underutilized VMs can be freed and turned off. Dynamically scaling up or down the cluster is possible by using elastic Cloud services offered by Amazon AWS or Azure. [25] is a model-driven autoscaler for Hadoop clus-

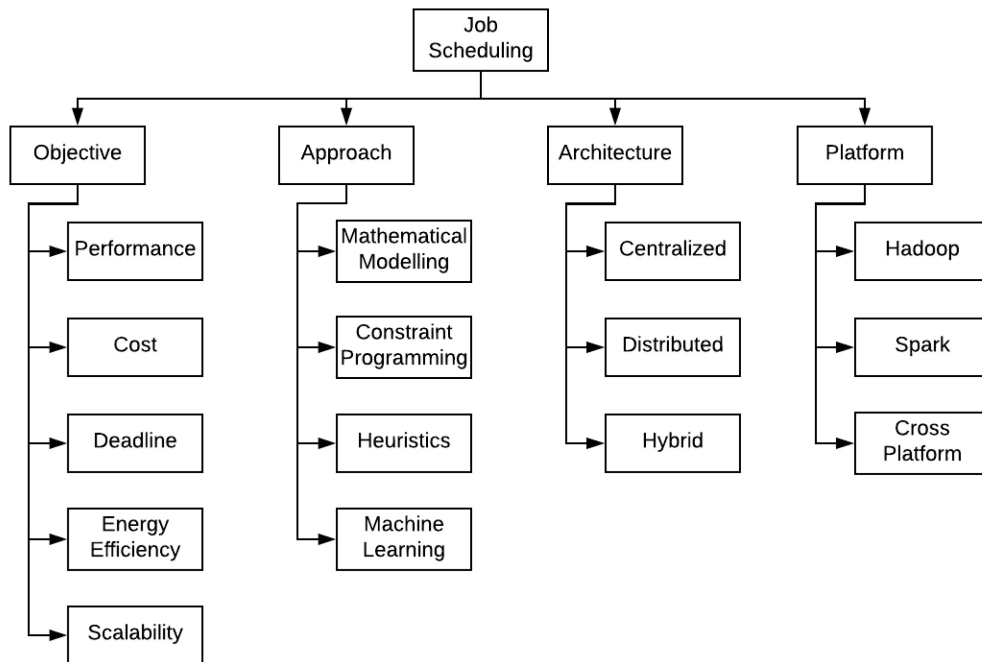


Figure 2.8: A Taxonomy of Scheduling of Big Data Applications on Cloud

ters. It uses novel gray-box performance models to predict job runtimes and resource requirements to dynamically scale the cluster so that SLA is satisfied.

2.4 A Taxonomy on Scheduling of Big Data Applications

Many types of research have been done in the task and resource scheduling in the Cloud computing environment. Researchers are trying to adapt existing scheduling approaches to facilitate the needs of big data applications. However, many challenges are posed due to the different characteristics of big data applications. In this section, different scheduling policies will be discussed. We have divided job scheduling approaches for big data applications based on four aspects. Figure 2.8 exhibits a taxonomy of big data job scheduling in the Cloud.

Based on the taxonomy, Table 2.1 shows a summary of comparison between the existing studies on Job scheduling for big data. In the following subsections, a detailed

comparison will be provided between all these works regarding the critical aspects of scheduling. When referring to a paper, we will follow the serial number of the corresponding paper from Table 2.1.

2.4.1 Objective

The target of a scheduling algorithm to achieve is called the objective. A scheduler can be single-objective or multi-objective, and it depends on the application scenario. Most of the scheduling algorithms focus on improving application performance. Besides, monetary cost reduction, handling soft or tight deadlines of jobs, energy-efficient placement of jobs and scalability of the overall system are also important objectives. Generally, the more objective is added to a scheduler, the more complex the decision-making progress becomes. Sometimes, the overhead of the scheduling solution could be a bigger issue rather than achieving the objects. Therefore, in real systems, different trade-offs are made on the objectives to design fast schedulers with fewer overheads.

Now, each of the following subsections will provide a detailed study on the existing literature from the perspective of the scheduling objectives.

Performance-oriented Scheduling

Performance improvement in scheduling can be achieved from two levels. First one is from the application/job level; where the target is to minimize the execution time of a job. The second one is from the cluster level; where a cluster scheduler has a global goal to improve the performance of the whole cluster. The most optimized way of scheduling is doing both. First, the job performance can be modeled by building mathematical models, machine-learning models, using monitoring data, etc. to set the appropriate resource requirement and configuration parameters for a job which is enough to maintain its SLA. Then, while each job is submitted, the cluster level scheduler improves the performance of the job by various techniques such as task consolidations in the same node to reduce network transfers, placing tasks close to data, order jobs based on their priority or deadline, etc.

Cost-efficient Scheduling

The monetary cost of running a big data processing cluster in a Cloud environment is crucial. Improper resource selection and resource scheduling might lead to resource wastage which intern increases the monetary cost of the cluster. If using VMs as the worker nodes of a big data cluster, it is often useful to turn-off unused or underutilized VMs to save cost if it does not affect performance/SLA of the jobs. Saving cost is mostly comes with a sacrifice of performance guarantee as cost can be saved by using a smaller number of resources in a cluster which might impact performance. Therefore, when both cost and performance is considered, resources are saved/consolidated only after ensuring a satisfying performance for all the jobs. In extremely scalable or fast scheduling systems, improving performance is the only goal and cost saving is mostly ignored.

Some jobs are associated with deadlines, and some job is time-critical or real-time and needs to be scheduled as soon as they arrive. Therefore, the deadline is an SLA parameter, and many schedulers try to minimize deadline violations. There are several techniques to achieve this first, the pre-emption mechanism where non-priority jobs are killed when priority jobs need to be scheduled. Second, reserving some resources that can be dedicated to time-critical or deadline-constrained jobs only. Lastly, ordering the jobs beforehand based on their deadlines. However, maintain the job deadline while handling other SLA constraints for jobs is difficult due to the presence of stragglers (large periodic jobs that might hold a considerable chunk of resources), job inter-dependency (a deadline-constrained job might wait for other critical or non-priority jobs), etc. When multiple objects such as cost, deadline and performance are considered together, generally there are strict priorities between the objectives. For example, the first objective is always ensuring a satisfiable performance of a job so that it meets its given deadline. When these objectives are satisfied, only then cost-saving is considered.

Energy-efficient Scheduling

One of the significant challenges of running big data applications in Cloud deployed cluster is minimizing their energy costs. Electricity used in the data centers in the USA accounted for about 2% of the total electricity usage of the whole country in 2010. Fur-

thermore, each year, the energy consumption by data-centers is increasing at over 15%. Lastly, the energy costs can take up to 42% of a data-centers total operational cost. It is predicted by IDC (Internet Data Corporation) that by the year 2020, big data analytics market will surpass $200 billion. Therefore, more and more data-centers are made, and these data-centers will consume a tremendous amount of energy soon. Consequently, it is crucial to make the scheduling techniques energy-efficient from both the application and the cluster side. Furthermore, from both the cluster and application side, consolidating resources to save cost leads to energy saving as it helps to reduce the number of active physical machines from the infrastructure side.

Scalable Scheduling

Scalable scheduling means that the resource management or scheduling algorithms are scalable to large clusters and can perform in the presence of high number of scheduling requests in a heterogeneous environment. Although the centralized approach of scheduling is less complicated to handle the complex steps of scheduling at one place, it is not as scalable as a distributed/hybrid approach of scheduling. It can be observed that scheduler scalability is addressed in only a few works (2, 3, 4, 6, 11, 28, 30) which mostly have distributed/hybrid architecture. However, as the existing cluster systems are growing massively on size and scale to handle massive amounts of analytics demands, future research should focus on the distributed or hybrid deployment of schedulers to make them scalable.

2.4.2 Approach

The solution method towards the scheduling problem varies. Generally, a complete and sophisticated scheduler has separate performance prediction and resource assignment modules. The performance models are built from mathematical models to predict the runtime of a job, cost of running a job, deadline violation, etc. in advance which helps to make accurate scheduling decisions. Constraint programming-based approaches try to minimize or maximize an objective by satisfying the constraint parameters set by the job and the restrictions of resources on the cluster. However, for both resource as-

signment/allocation and scheduling, the optimization problem is always modeled as an NP-Hard problem. Therefore, even if exact algorithms or constraint solving approach can find optimal scheduling decisions, it is not feasible in most of the case and only applicable in small-scale clusters. In contrast, heuristics or meta-heuristics approaches are faster, less-complicated and provided acceptable near-optimal solutions and can be scalable to large clusters. Nowadays, machine learning approaches are also becoming popular to build sophisticated and intelligent schedulers.

2.4.3 Architecture

Some scheduling designs are centralized, some are distributed. Recently, some hybrid approaches have also been proposed which uses both a distributed or local scheduler and a global scheduler. Generally, there are two levels in scheduling. One is at the cluster manager level which manages and schedules all the jobs submitted from multiple users. Another one is on the application level that schedules the tasks of a job to the allocated resources by the cluster-level scheduler. A centralized cluster-level scheduler design is less complicated as it controls all the jobs. However, for a massive cluster, a centralized scheduler could be a single point of failure. This limitation is solved with either having backup master nodes with the cluster manager (using tools like ZooKeeper) or by designing a distributed scheduler where the worker nodes co-ordinate with each other to manage the tasks from different jobs.

2.4.4 Framework

Most of the researches have tried to design efficient scheduling algorithms for Hadoop MapReduce based clusters as it was the mostly used distributed data processing framework in the last decade. However, as Apache Spark, Apache Storm, etc. are becoming more popular and vastly replacing Hadoop these days, the researchers are focusing on these frameworks now to devise scheduling algorithms. Lastly, due to the popularity of the cluster managers that support multiple different big data frameworks at the same time (Apache Mesos), or container-based platforms (Docker, Kubernetes); research has been going on building cross-platform cluster-level schedulers that can work with a

Table 2.1: Comparison between the existing job scheduling algorithms for big data

SL No.	Literature	Objective	Approach	Architecture	Framework
1	[26]	Deadline, Cost, Energy-efficiency	ILP, Heuristic	Centralized	Cross-Platform
2	[27]	Performance, Scalability	Sampling, Late bind	Distributed	Spark
3	[28]	Performance, Scalability	Sampling, SRPT	Hybrid	Cross-Platform
4	[29]	Cost, Scalability	AKNN, Naive Bayes	Hybrid	Cross-Platform
5	[30]	Deadline, Cost	Greedy Heuristics, ILP	Centralized	Cross-Platform
6	[31]	Cost, Scalability	Prediction-based	Centralized	N/A
7	[32]	Deadline, Cost	Prune Tree, Greedy Heuristics	Centralized	Cross-Platform
8	[33]	Performance, Deadline	Constraint Programming	Centralized	Hadoop
9	[34]	Deadline, Energy-efficiency	EDF, Periodic-DVFS	Centralized	Spark
10	[35]	Performance	Genetic algorithm	Centralized	Hadoop
11	[36]	Performance, Scalability	Multiplexing	Hybrid	Hadoop
12	[37]	Performance	Reinforcement learning	Centralized	Hadoop
13	[38]	Performance, Deadline	Greedy, Negotiation	Centralized	Hadoop
14	[39]	Performance, Deadline, Cost	Pareto-Frontier	Centralized	Hadoop
15	[23]	Performance	Task pre-emption	Centralized	Hadoop
16	[40]	Performance, Cost, Deadline	Greedy Heuristics	Centralized	Hadoop
17	[21]	Performance, Cost, Deadline	Mathematical model, Prediction	Centralized	Spark
18	[41]	Performance	Reservation aware, Dependency-aware	Centralized	Cross-Platform
19	[42]	Performance, Deadline	Graph Modelling	Centralized	Hadoop
20	[43]	Cost, Energy-efficiency	Reinforcement Learning	Centralized	Cross-Platform
21	[44]	Performance, Energy-efficiency	Machine Learning Classifiers	Centralized	Hadoop
22	[45]	Performance, Cost	Evolutionary algorithm	Centralized	Cross-Platform
23	[46]	Performance, Energy-efficiency	Time-series prediction, DVFS	Centralized	Spark
24	[47]	Performance, Energy-efficiency	Power profiles	Centralized	Cross-Platform
25	[48]	Performance, Deadline	Interference-aware	Centralized	Hadoop
26	[49]	Deadline, Cost	Pricing List, Bin Packing	Centralized	Hadoop
27	[50]	Performance, Deadline, Scalability	Job Profiles, Task Packing Resource Re-provision	Centralized	Hadoop
28	[51]	Performance, Scalability	Slot Management, Speculative Execution	Centralized	Hadoop
29	[52]	Performance	Reinforcement Learning	Centralized	Cross-Platform
30	[53]	Performance, Scalability	Job Profiles, Slot reconfiguration	Centralized	Hadoop
31	[54]	Deadline	Greedy Heuristic	Centralized	Hadoop

cluster manager to effectively handle jobs from different frameworks.

To summarize, it is always a hard challenge to provide a general scheduling strategy for all types of big data applications. To design a sophisticated scheduling algorithm, the type of big data application needs to be detected. Furthermore, depending on the user SLAs, the objectives should be chosen carefully. Then after setting the priority between different objectives, a suitable scheduling strategy can be devised.

2.5 Summary

In this chapter, we have discussed the basics of Cloud computing, the emergence of big data, processing frameworks and tools used to handle big data applications and an overall view of resource management for big data applications in the Cloud. We have specifically focused on the job scheduling aspect of resource management and provided a detailed taxonomy of job scheduling for big data applications. Furthermore, we have discussed the relevant research in scheduling and showed comparisons of various approaches regarding different aspects of scheduling. Lastly, we have highlighted some new research directions that need to be investigated to cope with the advanced resource management requirements in the modern era.

Chapter 3

Deadline-based Cloud Resource Allocation for Big Data Applications

When an application is deployed in a Spark cluster, all the resources are allocated to it unless users manually set a limit on the available resources. In addition, it is not possible to impose any user-specific constraints and minimize the cost of running applications. In this chapter, we present dSpark, a lightweight, pluggable resource allocation framework for Apache Spark. In dSpark, we have modelled the application completion time with respect to the number of executors and application input/iteration. This model is further used in our proposed resource allocation model where a deadline-based, cost-efficient resource allocation scheme can be selected for any application. As opposed to the existing frameworks that focus more on modelling the number of VMs to use for an application, we have modelled both the application cost and completion time with respect to executors, hence providing a fine-grained resource allocation scheme. In addition, users do not need to specify any application types in dSpark.

3.1 Introduction

NOWadays, huge amount of data is generated from social media, mobile devices, IoT and many other emerging applications. Therefore, data processing and analytics have become really important in all the major domains, such as research, business and industry. Apache Spark [6] is one of the most prominent big data processing plat-

This chapter is derived from:

- **Muhammed Tawfiqul Islam**, Shanika Karunasekera, and Rajkumar Buyya, "dSpark: Deadline-based Resource Allocation for Big Data Applications in Apache Spark", Proceedings of the 13th IEEE International Conference on e-Science (e-Science 2017), Auckland, New Zealand, October 24-27, 2017.

forms. It is an open source, general purpose, large-scale data processing framework. It mainly focuses on high speed cluster computing and provides extensible and interactive analysis through high level APIs. Spark can perform batch or stream data analytics, machine learning and graph processing. It can also access diverse data sources like HDFS [11], HBase [14], Cassandra [15] etc. and use Resilient Distributed Dataset (RDD) [16] for data abstraction. Spark runs programs faster than Hadoop-MapReduce [3] by performing most of the computations in-memory. In addition, it caches intermediate results in memory for faster re-processing of data. Spark can run locally in a single desktop, in a local cluster and on the Cloud. It runs on top of Hadoop Yarn [12], Apache Mesos [10] and the default standalone cluster manager.

In a Spark cluster, there are one or more worker nodes with the available resources (CPU cores, memory and disk). In addition, there is a master node which is responsible for allocating these resources to the applications. Each application uses the allocated resources to create executor processes where it can run tasks in parallel. Resource allocation in a Spark cluster can be done through the following three mechanisms: (1) *Default Resource Allocation*. It is used when the applications are submitted in a Spark cluster without specifying any resource allocation details. In this approach, all the applications will run in a FIFO style and each application consumes all the worker nodes. Hence, applications run one after another and when an application is running, it will use up all the worker nodes to create executors. (2) *Static Resource Allocation*. When an application is submitted, the user specifies how many executors, cores, memory etc. an application can have. Therefore, resources can be shared among multiple applications from one or more users. (3) *Dynamic Resource Allocation*. If this mode is turned on, applications may release idle executors to give back some resources to the cluster which can also be taken back in future if needed.

However, there are three major problems in these resource allocation mechanisms. First, when a single application is running in the cluster with the default resource allocation mechanism, it will consume all the resources. As a consequence, resource sharing among applications will be prevented. Second, in static resource allocation, the user has to manually set the amount of resources each application is going to use. Even with dynamic resource allocation, the user still has to set the initial amount of resources. As a

result, improper allocation of resources might lead to severe performance issues. Lastly, if a production cluster has user-specific deadlines, default resource allocation mechanism may not work since any application with a strict deadline might have to wait in the FIFO queue. Furthermore, inappropriate resource allocation in both static and dynamic resource allocation techniques might affect the deadlines.

In this chapter, we propose a resource allocation framework for distributed batch-based applications in Apache Spark. In addition, we propose an application completion time prediction model which can be built from the application profiles. This model is further used in the resource allocation model to select a deadline-based, cost-effective resource allocation scheme.

The main **contributions** of this work are as follows:

- We design an automatic, light-weight, pluggable **dSPark** *resource allocation framework* for Apache Spark that works from the master node along with the underlying cluster manager.
- We propose a *resource allocation model* where a cost-effective, deadline-based Resource Allocation Scheme (RAS) can be found for an application.
- We propose a *model* that predicts the completion time of an application based on the number of executors and properties of the application.
- We develop a *Spark Profiler* to profile any application with respect to varying input workloads, iterations, resource allocation schemes etc.
- We propose a simple *algorithm to generate Resource Allocation Schemes (RAS)* which can be used to deploy applications in an Apache Spark cluster.
- We implement the framework using the proposed models and algorithms. In addition, we run comprehensive experiments to show the accuracy and performance benefits of our proposed models.

The rest of the chapter is organized as follows. In section 3.2, we discuss the background of Apache Spark. In section 3.3, we describe the existing works related to this

chapter. In section 3.4, we formulate the resource allocation and application completion time prediction models. In section 3.5, we illustrate the architecture of the proposed dSpark framework. In section 3.6, we evaluate the performance of our proposed models. Section 3.7 concludes the chapter.

3.2 Background

As compared to the disk-based MapReduce tasks of a typical Hadoop system, Apache Spark allows most of the computations to be performed in memory and provides better performance for some applications such as iterative algorithms. The intermediate results are written to the disk only when it cannot be fitted into the memory.

Fig. 3.1 shows a typical Apache Spark cluster. Applications are submitted through a cluster manager to run in the cluster. Spark supports *Apache Mesos* or *Hadoop Yarn* as cluster managers to allocate resources among applications. In addition, its own default *Standalone* cluster manager is also sufficient to handle a production cluster. All these cluster managers support both static and dynamic allocation of resources. In static resource allocation, each application is deployed with a fixed amount of resources which cannot be changed during the life-cycle of that application. However, in dynamic resource allocation, idle resources can be released to the cluster and any other application can use them. These resources can also be taken back from the cluster in future if needed.

Workers are the physical/compute nodes of an Apache Spark cluster where one or more application processes can be created depending on the resource capacity. In Cloud deployments, one or more worker nodes can be created inside each allocated Virtual Machines (VM). A Spark cluster can have one or more worker nodes but there is only a single *Master* node that is responsible for managing the worker nodes.

Each application in Spark has a *SparkContext* object in its main program (also called the *Driver Program*) which creates and maintains *Executor* processes on worker nodes. An application uses its own set of executors to run tasks in parallel, in multiple threads and to keep data in memory and storage. In addition, these executors live for the whole duration of that application. All the executors of the same application must be identical in size. Hence, they will have same amount of resources (CPU cores, memory, disk).

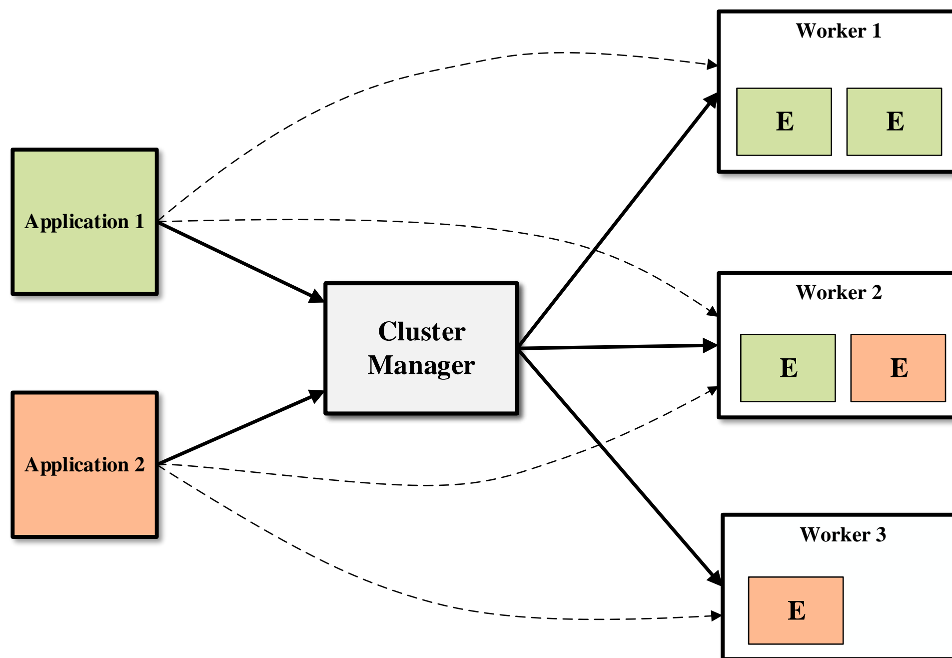


Figure 3.1: An Apache Spark Cluster

There are two benefits of isolating applications from each other. First, a driver program can independently schedule its own tasks in the acquired executors. Second, each worker can have multiple executors from different applications running in their own JVM processes.

Spark uses *Resilient Distributed Datasets (RDD)* to hold data in a fault tolerant way. Each job/application is divided into multiple sets of tasks called stages which are interdependant. All these stages form a directed acyclic graph (DAG) and each stage is executed one after another.

3.3 Related Work

A vast amount of research has been done in application performance modelling, resource provisioning and scheduling in Cloud-based systems. Here, we only focus on discussing the related research works done for big data processing platforms. Most of the research were done for MapReduce-based big data frameworks as it was the most popular big data processing paradigm in the last decade. ARIA [55] was designed for

MapReduce based environments where job profiles of map and reduce tasks of an application are collected to build job profiles. A MapReduce performance model is built from the job profiles which is used to estimate the required resources for a job completion given its Service Level Objective (deadline). In [18], performance modelling of MapReduce jobs was done in heterogeneous Cloud environments. [56] proposed a resource provisioning framework for MapReduce workloads. [57] showed a deadline-based workload management for MapReduce workloads. [58] predicts the expected performance of a big data workload from historical performance results using Support Vector Machine (SVM). However, these approaches are not straight forward to apply in Apache Spark as it is a DAG-based in-memory analytics platform.

[59] evaluated the performance of Apache Spark in MareNostrum supercomputer to explore its efficiency and applicability in HPC setup. In addition, they have also developed a framework called Spark4MN to automate the use of a Spark cluster in HPC environment. Furthermore, they have explored the impacts on performance by differernt parameters like worker size, tasks per core etc. Similar to this work, [60] also investigated different configuration parameter tuning of Spark applications. They have identified a set of important parameters and provided a trial-and-error based methodology to tune these parameters for performance speed-ups. A machine learning based configuration parameter tuning approach is proposed in [61]. Their method is composed of binary classification and multi-classification. As Spark has a huge parameter space, they have taken a random sample from the parameter space and generated a parameter list of 500 records for each type of workload. Then real execution data on these parameter lists is taken to train their models. Their experimental results show that a Decision Tree (C5.0) provides good accuracy and performance in diverse workloads.

[62] investigated the problem of resource waste that occurs while a Spark application runs in all the nodes in a cluster. To address this problem, they have proposed dynamic partitioning based solutions that tune the degree of parallelism of Spark application during execution to reduce resource consumption. To achive this, they have to trade small amount of running time. [63] built multiple polynomial regression models on the application profile data and applied k-fold cross validation to choose the best model to predict application execution time with unknown input data set or cluster configura-

Table 3.1: Related Work

Parameter	Related Work						dSpark
	[60]	[61]	[62]	[63]	[19]	[21]	
Framework	✗	✗	✗	✗	✗	✗	✓
Performance Modelling	✓	✓	✓	✓	✓	✓	✓
Executor Cost Modelling	✗	✗	✗	✗	✗	✗	✓
Deadline	✗	✗	✗	✗	✗	✓	✓
Cost Saving	✗	✗	✗	✗	✗	✓	✓
Resource Saving	✗	✗	✗	✗	✗	✗	✓

tion. [19] tried to model application performance in DAG-based in-memory analytics platforms and they have used Apache Spark to validate their methods. In this work, the execution times from different stage of an application is collected and then used to predict the execution time of the application. However, these works did not consider cost minimization and user-specific deadlines. In optEx [21], a deadline oriented cost optimization model was proposed. However, in optEx, the user needs to specify the type of the application before deployment.

In all these works related to Spark performance modelling, they have considered VMs as the unit of resource of an application and tried to estimate application completion time with different number of VMs. However, in dSpark, we have considered executor processes as a unit of resource for the application. For any size of machine either in local or Cloud deployed cluster, our model is capable of finding more fine-grained resource allocation schemes. Therefore, multiple applications will be able to run executor processes in the same worker node depending on the worker and executor size. Furthermore, dSpark also utilizes a flexible cost model that can be customized to integrate user's own pricing policies. Lastly, dSpark can provide efficient resource allocation schemes under varying SLO deadlines. The summary of the comparison between our work and other closely related works is given in Table 3.1.

Table 3.2: Definition of Symbols

Symbol	Definition
A	a Spark application
E	total number of executors
P_{vm}	price (per second) of a VM
M_e	memory (GB) in each executor
C_e	number of cores assigned to each executor
P_e	price of one executor
N_w	total number of workers in the cluster
C_w	total number of cores in each worker
M_w	total memory (GB) in each worker
E_{max}	maximum possible executors in the cluster
T	completion time of an application
D	deadline of an application
I	input size or iteration of an application
RAS	resource allocation scheme
$RASL$	list of resource allocation schemes

3.4 Problem Formulation

3.4.1 Cost-efficient Resource Allocation Model

An Apache Spark cluster comprising of master and worker nodes can be deployed on Cloud Virtual Machines (VM). For simplicity of our proposed model, we assume that all the VMs used as worker nodes are homogeneous. Therefore, each of the VM will have same amount of CPU cores, memory and storage disk. To deploy an application in the cluster, a Resource Allocation Scheme (RAS) needs to be defined. In each RAS, the total number of executors, CPU cores in each executor and memory in each executor should be specified. Our goal is to choose a cost-efficient RAS which ensures that an application will be completed before the user-specified deadline.

Suppose, we have an Apache Spark cluster with N_w total number of worker nodes and one (1) master node. In addition, all these nodes are created in distinct VMs. As

all the workers are homogeneous, each worker has C_w CPU cores and M_w total memory. Furthermore, the price of running each VM is P_{vm} (\$) per second. For a particular application (A), the user specifies a deadline (D) before which this application needs to complete. In addition, the user also defines the cores per executor (C_e) value. If all the memory of a worker (M_w) is evenly associated among all the cores (C_w), then (M_w/C_w) amount of memory will be associated with each core. Therefore, memory in each executor (M_e) will be $C_e * (M_w/C_w)$. In Apache Spark, for a particular application, all the executors need to be identical. Therefore, our problem is now to find the number of executors (E) to use with an application that meets the user deadline (D) and also minimizes the total cost.

We model this problem as a constrained non-linear optimization problem as follows:

$$\text{Minimize: } Cost = P_e * E * T \tag{3.1}$$

subject to:

$$1 \leq E \leq E_{\max} \tag{3.2}$$

$$T \leq D \tag{3.3}$$

where:

$$P_e = C_e * (P_{\text{vm}}/C_w) \tag{3.4}$$

$$M_e = C_e * (M_w/C_w) \tag{3.5}$$

$$E_{\max} = N_w * (C_w/C_e) \tag{3.6}$$

$$T = f(E, I) \tag{3.7}$$

$$E, C_e, M_e \in \mathbb{Z} \tag{3.8}$$

Cost Minimization: Eqn. 3.1 shows the objective function where $Cost$ is the dependent variable and executors (E) and application completion time (T) are the decision variables. In addition, P_e is a constant value which represents the cost of running one (1) executor.

Executor Capacity Constraint: As shown in Eqn. 3.2, we have a lower bound and an upper bound on the number of executors of an application. The lower bound should

be one (1) as each application needs at least 1 executor to process data and the upper bound (E_{max}) depends on the available cluster resources as shown in Eqn. 3.6.

Application Deadline Constraint: As shown in Eqn. 3.3, application completion time (T) of a selected configuration should meet the user specified deadline (D). In a case where the model finds multiple resource configurations that satisfy the deadline constraint, it will only select the one which has the lowest cost.

Executor Price Estimation: As we associate equal amount of memory with all the CPU cores in a VM, the number of used CPU cores represents the price of a VM. Therefore, we can find the price (per second) of a single CPU core from the actual VM price by dividing the price for running each VM (P_{vm} ($)) with total number of available cores in a VM (C_w). Eqn. 3.4 shows the price estimation function of an executor process. Pricing policy of this model can be easily converted to a different scenario and allow users to use their own VM pricing model.

Memory Capacity Constraint: The amount of memory for an executor (M_e) depends on the number of cores (C_e) in that executor. In addition, it is also capped by the total memory of a worker as shown in Eqn. 3.5.

Application Completion Time Prediction: As shown in Eqn. 3.7, the proposed optimization model finds the value of completion time (T) as a function of executor (E) and the total application input (I). We propose an application completion time prediction model to be used as this function. This model will be discussed in detail in the following subsection.

Integer Constraints: The number of executors (E), cores in each executor (C_e) and memory in each executor (M_e) must be integers as shown in Eqn. 3.8.

3.4.2 Application Completion Time Prediction Model

An Apache Spark application uses it's allocated executors to process multiple chunks/splits of the whole input in parallel. The partitioning or splitting of the input imposes a little overhead on the actual running time. In addition, after all the processing is finished, the result needs to be serialized which also adds up to the total execution time. If the number of input chunks is more than the number of executors, these input chunks are

processed like a batch in each executor. However, adding too many executors to achieve more parallelism can cause overheads due to serialization, de-serialization and intensive shuffle operations in the network. Therefore, when an application is given more and more executors, performance boost can be significant at the start. However, after some point, adding more executors does not give any performance benefit rather resources are wasted. Therefore, for a fixed input (I) of an application, we can assume that the relationship between executors (E) and completion time (T) can be modelled like a power function as:

$$T(E) = \alpha * E^{\beta} + \gamma \tag{3.9}$$

where α, β and γ are the power model coefficients.

However, in reality, the application input is not a fixed parameter. Therefore, we further assume that the coefficients in Eqn. 3.9 are determined by the application input (I) and can be modelled like a power function as:

$$\alpha(I) = u_{\alpha} * I^{v_{\alpha}} + w_{\alpha} \tag{3.10}$$

$$\beta(I) = u_{\beta} * I^{v_{\beta}} + w_{\beta} \tag{3.11}$$

$$\gamma(I) = u_{\gamma} * I^{v_{\gamma}} + w_{\gamma} \tag{3.12}$$

where, $\{u_{\alpha}, v_{\alpha}, w_{\alpha}\}$, $\{u_{\beta}, v_{\beta}, w_{\beta}\}$ and $\{u_{\gamma}, v_{\gamma}, w_{\gamma}\}$ are the power model coefficients in Eqn. 3.10 Eqn. 3.11 and Eqn. 3.12, respectively. If we substitute α, β and γ of Eqn. 3.9, we find:

$$T(E, I) = (u_{\alpha} * I^{v_{\alpha}} + w_{\alpha}) * E^{(u_{\beta} * I^{v_{\beta}} + w_{\beta})} + u_{\gamma} * I^{v_{\gamma}} + w_{\gamma} \tag{3.13}$$

Eqn. 3.13 establishes the relationship of application completion time (T) with respect to both executor (E) and application input or iteration (I). Hence, it can be used in the resource allocation model (Eqn. 3.7) to determine the completion time of an application. In addition, this model can predict application completion time with any size of input/iteration and any number of possible executors.

To determine the coefficients in Eqn. 3.13 for a particular application, we make n

observations of the application with different inputs $\{I_1 \ to \ I_n\}$. For each application input, we measure the T values with respect to different E values and fit them to establish a relationship as shown in Eqn. 4.6. Therefore, we will get three (3) set of coefficients: $\{\alpha_1 \ to \ \alpha_n\}$, $\{\beta_1 \ to \ \beta_n\}$, $\{\gamma_1 \ to \ \gamma_n\}$ for n observations. Now, if we fit $\{\alpha_1 \ to \ \alpha_n\}$ vs $\{I_1 \ to \ I_n\}$ values as Eqn. 3.10, we will find $\{u_\alpha, v_\alpha, w_\alpha\}$ coefficient values. Similarly, the values of coefficient sets $\{u_\beta, v_\beta, w_\beta\}$ and $\{u_\gamma, v_\gamma, w_\gamma\}$ can be found.

3.5 dSpark Framework Overview

A production cluster can be built with multiple computing nodes connected in a local area network (LAN). However, we can avoid the hassle of maintaining local machines by using Cloud services as it offers more affordable and flexible computing resources to deploy a cluster. dSpark framework can be used both locally or in a Cloud-deployed cluster.

Fig. 3.2 shows the proposed architecture of the *dSpark Framework*. We have two (2) main modules: *Profiler* and *Resource Allocator*. These modules work collaboratively on top of the cluster manager to generate a cost-effective, deadline-aware RAS for an application. This RAS can be used for real deployment of this application in the cluster.

Resource Allocator

It is the main component of our system. Algorithm 1 shows the steps performed by this module. As an input to this algorithm, the application program (A), input/iteration (I), user-specific deadline (D) and VM price (P_{vm}) is given. At first, the Profiler module is invoked to generate application profiles (line 3). Then the application completion time prediction model is built (line 4) as discussed in section III.B. While building the model, the algorithm finds all the coefficient values of Eqn. 4.10. The $Find - RAS()$ procedure called in line 5 implements the resource allocation model to select the optimal RAS.

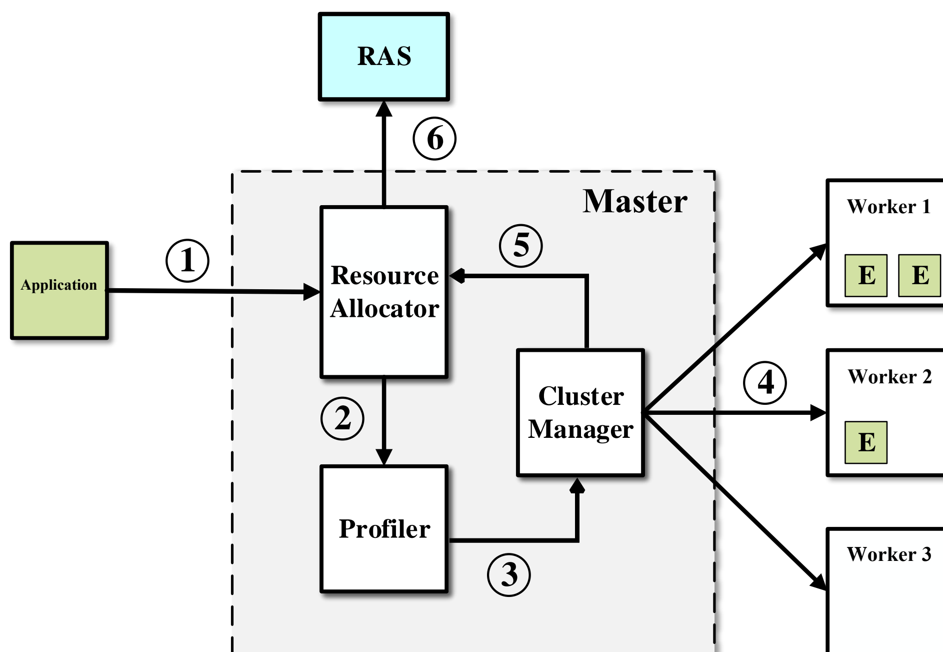


Figure 3.2: dSpark Architecture

Algorithm 1: Resource Allocator Algorithm

Input: A, I, D *and* P_{vm}
Output: Resource Allocation Scheme (RAS)
1 $ApplicationProfiles \leftarrow$ PROFILER(A, I)
2 TIME-ESTIMATE-MODEL$(ApplicationProfiles)$
3 $RAS \leftarrow$ FIND-RAS$(D, I, P_{\text{vm}}, \alpha, \beta, \gamma)$
4 **return** RAS

Spark-Profiler

This module is controlled by the Resource Allocator module to generate application profiles for an application. The profiler module runs the application with different RAS, varying inputs or iterations (in case of iterative applications like PageRank) in the cluster. After that, it uses a sub-module called *LogParser* to get the completion times of an application from the logs in the master node. Finally, it generates the application profiles and sends to the Resource Allocator module. Spark-Profiler is configurable by the user to set the portion of input that needs to be profiled before the actual deployment of an application. By default it uses 10% of the input workload for profiling. As the ap-

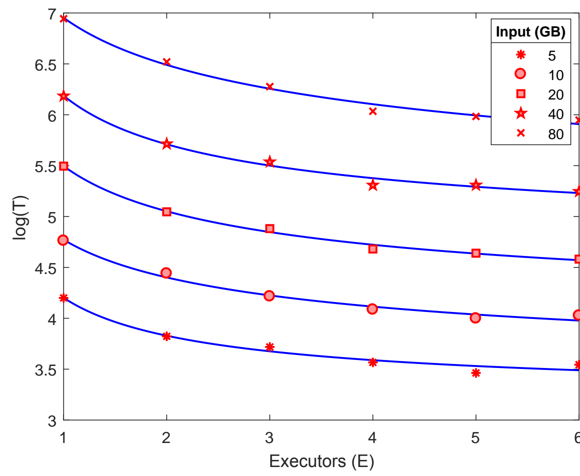
plication completion time prediction model uses multiple increasing input to build the model, we used the initial chunk repeatedly to increase it to the desired size.

Algorithm 2: Resource Allocation Scheme (RAS) Generation

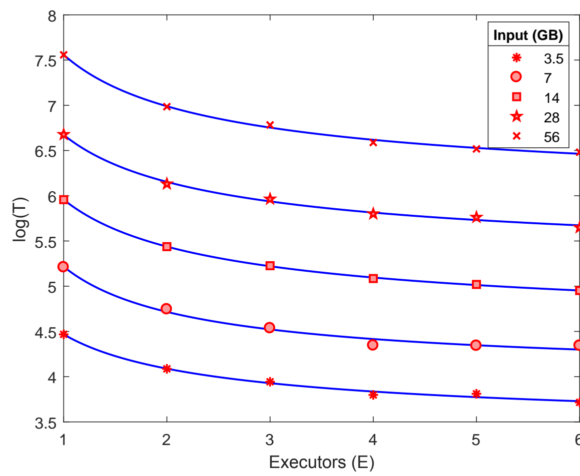
Input: N_w, M_w, C_w *and* C_e
Output: Resource Allocation Scheme List ($RASL$)
1 CALCULATE(M_e) ($Eqn.$ 3.5)
2 CALCULATE($E_{\max}$) ($Eqn.$ 3.6)
3 $E \leftarrow 1$
4 **while** $E \leq E_{max}$ **do**
5 $\quad RAS \leftarrow \{C_e, M_e, E\}$
6 $\quad RASL \leftarrow RASL + RAS$
7 $\quad E \leftarrow E + 1$
8 **return** $RASL$

To submit an application to a Spark cluster, a RAS need to be specified as a limit on the possible cores per executor (C_e), memory per executor (M_e) and total executors (E) per application. In order to generate the application profiles, we need to run the application with different RAS and input/iteration. Algorithm 2 shows a simple Resource Allocation Scheme (RAS) generation technique which is used by the Profiler module. To generate the possible resource allocation schemes, knowledge on the total amount of cluster resources is needed. As previously noted, we assume that all the worker nodes (VM from Cloud perspective) are homogeneous in a Spark cluster. Therefore, as an input to our algorithm, the total number of worker nodes and only the configuration of a single worker node is given. At first the algorithm finds M_e and $E_{\max}$ values. In the next part of the algorithm (line 5 to line 8), the total number of executors per application is varied to generate different RAS. All the generated RAS are added to a list called Resource Allocation Scheme List (RASL).

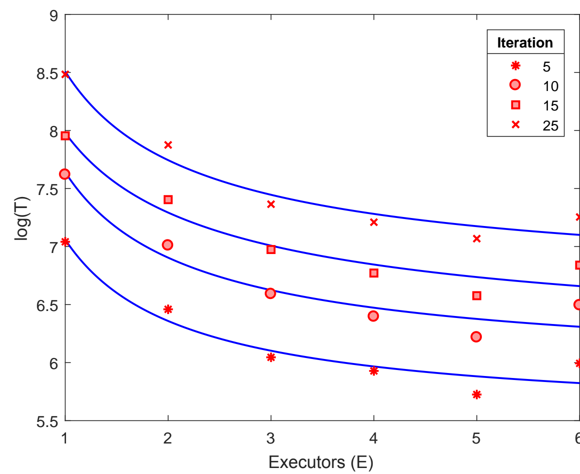
dSpark can be installed as a small plug-in to the master node of an Apache Spark cluster. First, the user needs to specify any required configurations in dSpark. Then, the user should submit the applications directly to dSpark instead of the cluster. After selecting the RAS for an application, dSpark automatically submits the application to the production cluster with the selected RAS.



(a) WordCount

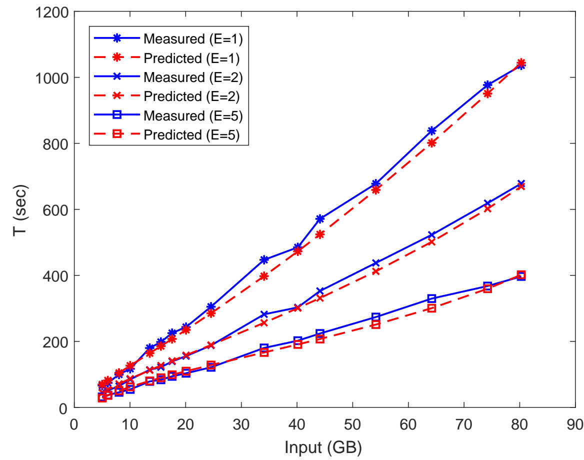


(b) Sort

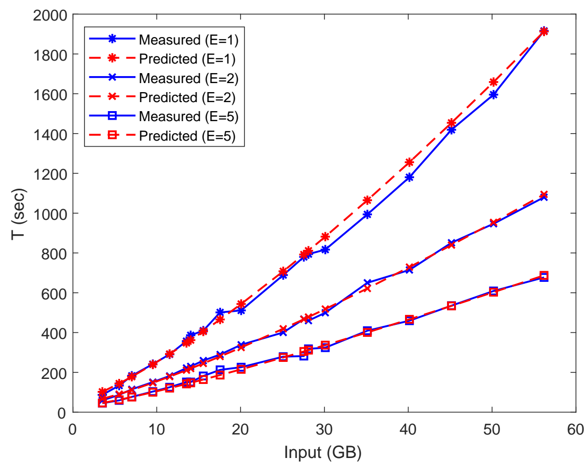


(c) PageRank

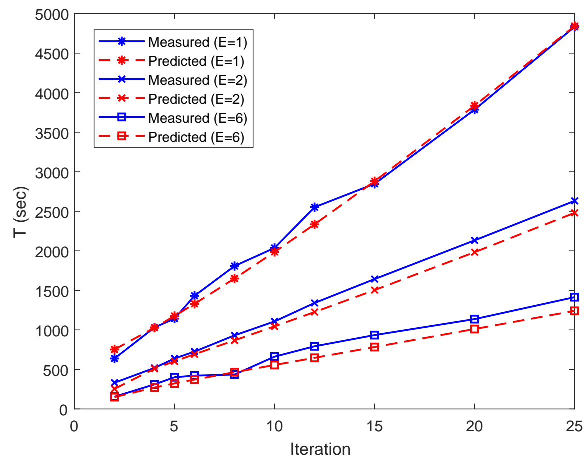
Figure 3.3: Application completion time modelling.



(a) WordCount



(b) Sort



(c) PageRank

Figure 3.4: Accuracy of application completion time prediction for different applications.

3.6 Performance Evaluation

3.6.1 Implementation

We have used Java programming language to develop the proposed framework. We have implemented the *Spark-Profiler* module to profile any spark application with a given input size and a RAS. This module uses SparkLauncher Java API [64] to submit applications to the cluster. After an application finishes its execution, a sub-module called LogParser is used to parse the logs in the master node to retrieve the completion time of that application. We have implemented *Resource Allocator* as a separate module and it controls the *Spark-Profiler* module. At first this module reads the configuration files to get the information about the cluster resources. As discussed in Algorithm 1, this module implements both *Application Completion Time Prediction Model* and the *Resource Allocation Model* as two different procedures. To build up the application completion time prediction model, we have applied curve-fitting tools from Apache Commons Maths Library [65]. For solving the constrained minimization problem in our resource allocation model, we haved used JOptimizer Library [66].

3.6.2 Experimental Setup

Cluster Configuration

We have deployed an experimental Apache Spark cluster on Microsoft Azure Virtual Machines (VM). For the master node, we have chosen "standard D4" size VM instance which has 8 cores and 28 GB memory. We have made two (2) worker nodes with "standard D5v2" size VM instance each having 16 cores and 56 GB memory. For storage, we have created an Azure Storage Account to deploy a shared storage device mounted in all the VMs. The replication option chosen for this storage was "Locally redundant storage (LRS)". In LRS, data is replicated three times within a single data center which is located in a single region. All the volumes and VMs were created in the "Australia South-East" region. We have installed Ubuntu Server Version 16.04 LTS in all the nodes and installed Apache Spark Version 2.0.1 on top of it. In addition, we have utilized the

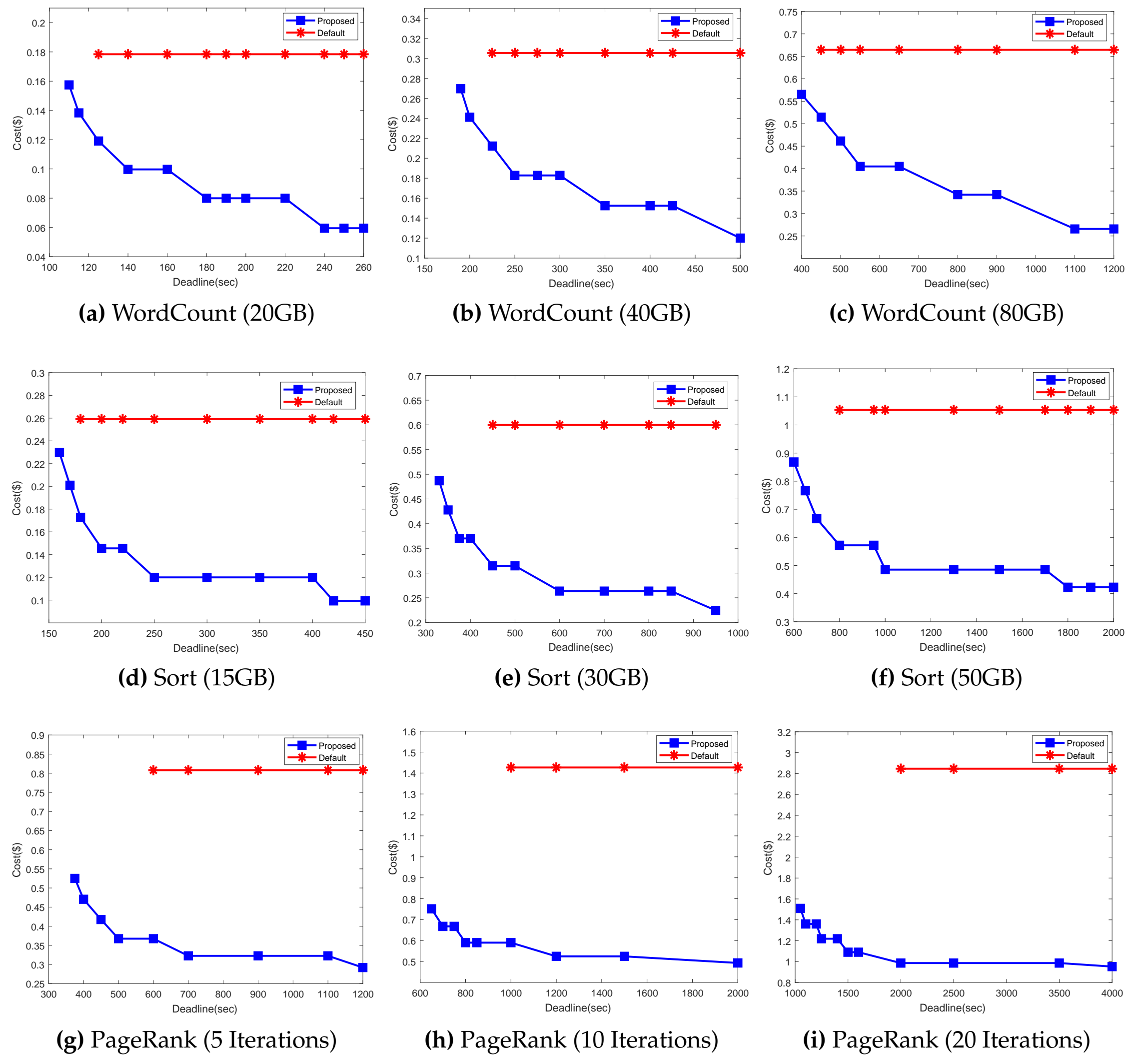
standalone cluster manager that comes by default with Apache Spark. We have kept 15 CPU cores and 45 GB of memory of a VM for each worker node. For OS specific daemons and other application programs, we have left the rest of the CPU cores and memory. In our experiments, we have defined the C_e value to be five (5) which is recommended by the Spark developers because using large number of cores in a single executor results bad I/O throughput and having more executors each with fewer cores results in high garbage collection (GC) and scheduling overhead. However, this value can be configured in dSpark by the user if required. The price (P_{vm}) of each "standard D5v2" instance was $0.0795 AUD at the time of the experiments.

Benchmarking Applications

We have used BigDataBench [67], a big data benchmarking suite to evaluate the performance of our proposed models. We have chosen three different types of applications. These are: (1) *WordCount:* compute intensive application, (2) *Sort:* memory and compute intensive application and (3) *PageRank:* iteration based shuffle intensive application.

Application Profiles

We have used the Spark-Profiler module to collect application profiles for all the benchmarking applications. For WordCount application, we have collected application profiles for 5 GB, 10 GB, 20 GB, 40 GB and 80 GB of input workloads. For Sort application we have collected application profiles for 3.5 GB, 7 GB, 14 GB, 28 GB and 56 GB of input workloads. Lastly, for PageRank application, we have collected application profiles for 5, 10, 15, 20 and 25 iterations for the same 4 GB input graph. We have built the application completion time prediction model as discussed in section III.B and calculated all the coefficients of Eqn. 4.10.



(a) WordCount (20GB) **(b)** WordCount (40GB) **(c)** WordCount (80GB)

(d) Sort (15GB) **(e)** Sort (30GB) **(f)** Sort (50GB)

(g) PageRank (5 Iterations) **(h)** PageRank (10 Iterations) **(i)** PageRank (20 Iterations)

Figure 3.5: Cost of resource usages by the proposed and the default approach for different applications.

3.6.3 Analysis of Results

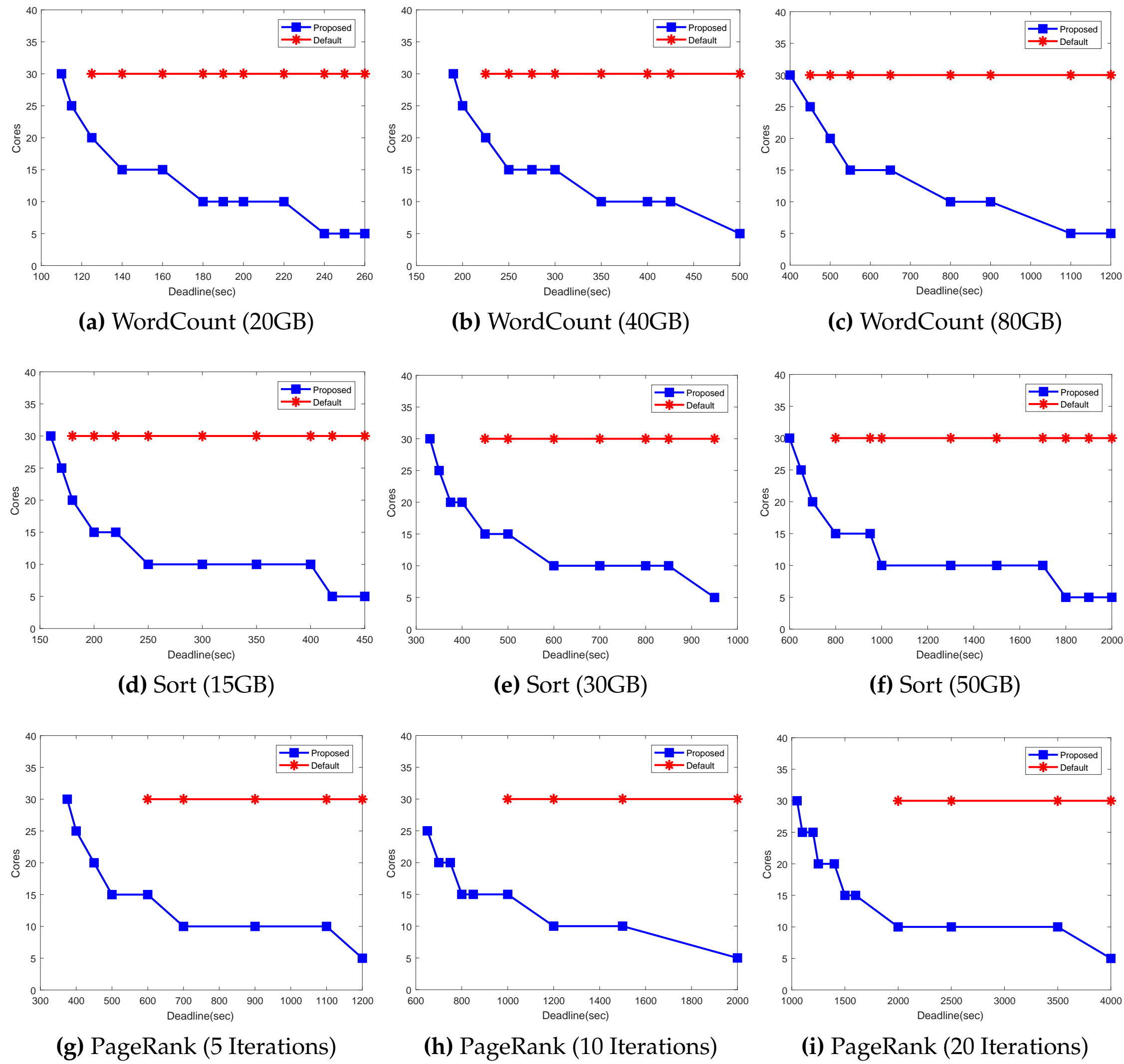
Accuracy of Application Completion Time Prediction

Fig. 3.3 shows E vs T curves for three (3) different applications: WordCount (3.3a), Sort (3.3b) and PageRank (3.3c). For WordCount and Sort, we have used different size of application inputs. For PageRank application, we have considered different iterations on the same input graph. From these graphs, it can be observed that there is a decrease in execution time when the number of executors is increased. However, the decrease in execution time is steeper upto 3 or 4 executors. After this point, the execution time does not decrease significantly even if more executors are used for an application. Due to some performance limiting factors like: data serialization/de-serialization, network I/O and shuffle operations, this behaviour was seen from the applications. As all the curves shown in Fig. 3.3 follows a steady power model, it validates our assumption of using power models to establish the relationship between executor (E) and application completion time (T). While we built up the application completion time prediction model, we have found steady power models for the input vs coefficient graphs.

Fig. 3.4 illustrates the difference between the predicted completion times and the measured completion times of three (3) different applications. From all these graphs, it can be clearly seen that the predicted completion time curves fall closely to the measured completion time curves. We have computed the relative error $RE = (T_{predicted} - T_{measured})/T_{measured}$. We got a mean RE of 5%, 3% and 8% for WordCount, Sort and PageRank applications respectively. As our proposed model has a lower mean RE values for all the experimented applications, it can be used with the resource allocation model.

Cost Analysis

Fig. 3.4 compares the cost of running applications between the proposed resource allocation model and the default Spark resource allocation. We have measured the cost for both approaches with various user-specific deadlines. Fig. (3.5a-3.5c) illustrates cost comparison of the WordCount application with 20 GB, 40 GB and 80 GB inputs respec-



(a) WordCount (20GB) **(b)** WordCount (40GB) **(c)** WordCount (80GB)

(d) Sort (15GB) **(e)** Sort (30GB) **(f)** Sort (50GB)

(g) PageRank (5 Iterations) **(h)** PageRank (10 Iterations) **(i)** PageRank (20 Iterations)

Figure 3.6: Comparison of resource usages between the proposed and the default approach

tively. Fig. (3.5d-3.5f) illustrate cost comparison of the Sort application with 15 GB, 30 GB and 50 GB inputs respectively. Lastly, Fig. (3.5g-3.5i) illustrate cost comparison of the PageRank application with 5, 10 and 20 iterations respectively. As seen from all these graphs, cost for running application in the default approach shows a horizontal line in the all cases. In the default approach, each application uses all the resources in the whole cluster. Therefore, even for various user-specified deadlines, it gives the cost of using all the resources. However, our proposed model tends to use more resources only when an application has a strict deadline. In this case, the model utilizes more resources to meet the deadline thus costs higher. When the deadline starts to become more flexible, our model uses a small set of resources to meet the deadline and reduces the cost significantly.

Resource Usage Analysis

Fig. 3.6 compares the resource usage between the proposed approach and the default approach. In our models, we have considered Executors (E) as a chunk of resource as the actual VM resources (CPU cores, memory) are distributed among the executors. However, the size of the executors used by both of these approaches are not the same. In default resource allocation technique, only one (1) executor is launched in each worker node. Therefore, in our experimented cluster, the default approach makes two (2) executor each having fifteen (15) CPU cores. However, in our proposed approach, the cores per executor (C_e) value is flexible and can be tuned according to the application needs. As mentioned before, in our experimental setup, a developer recommended value is used for (C_e). Therefore, we compare the default and proposed approach in terms of CPU cores usage per application. Memory consumption is not shown because we evenly associated all the memory in a VM with the CPU cores. Therefore, higher number of CPU cores usage reflects high amount memory usage. Fig. (3.6a-3.6c) compares CPU cores usage of WordCount application for different size of input workloads. In addition, Fig. (3.6d-3.6f) compares CPU cores usage of Sort application for different size of input workload. Lastly, Fig. (3.6g-3.6i) compares CPU cores usage of PageRank application for different iterations of the same input graph. It can be observed from these graphs that,

in all cases, default approach uses all the CPU cores available in the whole cluster to run an application. As we have total 30 CPU cores in the whole cluster, in default approach, all the applications have used 30 CPU cores. Variations in the user-specific deadline does not change resource usages for default resource allocation. However, in the proposed approach, our resource allocation model tries to meet user-specific deadline for an application. If it is possible to use less resources to meet the user-specific deadline, to minimize cost and resource usages, our model selects that resource allocation scheme. Therefore, for the applications with strict deadlines, we observe high resource usages and for the applications flexible deadlines, we see less resource usages.

From both Fig. 3.5 and Fig. 3.6 it can be observed that our model handles both strict and flexible deadlines better than the default approach. As we discussed before, using large executors poses performance overheads on the applications. Therefore, the default approach shows poor performance and more deadline violation occurs with strict deadlines. In both of our analysis, we did not include the initial profiling cost as it needs to be done only once for each application.

3.7 Summary

Distributed, large-scale processing of big data has a significant impact on both research and industry. Apache Spark is becoming more popular as a cluster computing engine due to its high-speed data processing capability, extensive applicability in various domains and wide-range of high level APIs. To support user-specific SLA requirements and to maximize an Apache Spark cluster utilization, our research focuses on proposing a cost-effective resource allocation model. The aim is to allow the user a way of automatic and efficient deployments of applications in a local or Cloud cluster. We have developed a profiler for Spark which can be used to profile an application in the real cluster in terms of different resource allocation schemes and input workloads. Moreover, we have developed a light-weight resource allocation framework called dSpark that can be plugged into the master node of an Apache Spark cluster. Applications can be submitted to dSpark instead of directly submitting to the cluster. Based on the application profiles received from the profiler, dSpark uses the proposed resource allocation

model to select a deadline-based cost-effective resource allocation scheme to deploy an application to the cluster.

We have conducted experiments to evaluate the efficiency of our proposed models. In addition, we have shown the accuracy of the application completion time prediction model for three (3) different applications. The mean relative error in the prediction model was less than 7% for different types of applications. Furthermore, we have evaluated the effectiveness of the resource allocation model in terms of cost and resource usage and compared the results with the default resource allocation approach in Spark. We have showed that our model selects cost-effective resource allocation schemes that effectively handles various user-specific deadlines. In addition, unlike some existing works, dSpark does not require the users to specify application types as it would be difficult for an end-user to have proper understanding of the application to determine it's type.

As our application completion time prediction model is built by using knowledge from the application profiles, the accuracy of this model depends on the intensity of application profiling. The accuracy of this model increases with a higher number of application profiles. Therefore, there is a clear trade-off between model accuracy and the level of profiling. However, application profiles can be made from past application runs to reduce profiling overhead.

Chapter 4

Scheduling Big Data Applications in Cloud Computing Environments

Job scheduling is one of the most crucial components in managing resources, and efficient execution of big data applications. Specifically, scheduling jobs in a cloud-deployed cluster are challenging as the cloud offers different types of Virtual Machines (VMs) and jobs can be heterogeneous. The existing works in cluster scheduling mainly focus on improving job performance and do not leverage from VM types on the cloud to reduce cost. In this chapter, we propose efficient scheduling algorithms that reduce the cost of resource usage in a cloud-deployed Apache Spark cluster. In addition, the proposed algorithms can also prioritize jobs based on their given deadlines. Besides, the proposed scheduling algorithms are online and adaptive to cluster changes.

4.1 Introduction

BIG Data processing has become crucial due to massive analytics demands in all the major business and scientific domains such as banking, fraud detection, healthcare, demand forecasting, and scientific explorations. Data processing frameworks such as Hadoop, Storm, and Spark [6] are the most common choice when it comes to big data processing. Large organisations generally run private compute clusters with one or more big data processing frameworks on top of it. As public cloud services can pro-

This chapter is derived from:

- **Muhammed Tawfiqul Islam**, Satish N. Srirama, Shanika Karunasekera, and Rajkumar Buyya, "Cost-efficient Dynamic Scheduling of Big Data Applications in Apache Spark on Cloud", *Journal of Systems and Software (JSS)*, Volume 162, Pages: 1-14, ISSN: 0164-1212, Elsevier Press, Amsterdam, The Netherlands, April 2020.

http://hadoop.apache.org/
http://storm.apache.org/

vide infrastructure, platform, and software for storing and computing of data, it is also becoming popular to deploy the big data processing clusters on public clouds. However, scheduling these big data jobs can be difficult in a cloud-deployed cluster since the jobs can be of different types such as CPU-intensive, memory-intensive, and network-intensive. Furthermore, jobs can also vary based on their resource demands to maintain a stable performance. Moreover, various types of Virtual Machines (VM) instances available on the cloud make it difficult to generate cost-effective scheduling plans. Therefore, in this chapter, we propose efficient job scheduling algorithms that reduce the cost of using a cloud-deployed Apache Spark cluster while enhancing job performance.

To demonstrate the effectiveness of our scheduling algorithms, we have chosen Apache Spark as our target framework because it is a versatile, scalable and efficient big data processing framework and is rapidly replacing traditional Hadoop-based platforms used in the industry. Spark utilizes in-memory caching to speed up the processing of applications. The resource requirements of a Spark job can be specified by using the number of required executors for that particular job, where each executor can be thought of as a process, having a fixed chunk of resources (e.g., CPU, memory and disk). However, different jobs can have varying executor size requirements depending on the type of workloads they are processing. Therefore, jobs exhibit different characteristics regarding resource dependability.

The default scheduling mechanism for Spark job scheduling is *First in First Out (FIFO)*, where each job is scheduled one after another. If no resource limit is set, one job might consume all the resources in the cluster. On the other hand, if the user sets a limit on the required resources of a job, the remaining resources can be used to schedule the next job in the queue. In addition to the FIFO scheduler, a *Fair Scheduler* is also available to prevent resource contention among jobs. By default, both of these schedulers place the executors of a job in a round-robin fashion in all the VMs/worker nodes for load-balancing and performance improvement. However, when a cloud-deployed cluster is not fully loaded with jobs, round-robin executor placement leads to resource wastage in all the VM. Although Spark also has an option to consolidate the executor placements,

https://spark.apache.org/docs/latest/job-scheduling.html#scheduling-across-applications

the cluster manager does not consider the resource capacity and price of different cloud VM instance types, and thus fails to make cost-efficient placement decisions. Most of the existing scheduling techniques focus on Hadoop-based platforms [54, 68, 69, 70]. Nevertheless, these mechanisms cannot be directly applied to Spark job scheduling as the architectural paradigm is different from in-memory computing frameworks. A very few works have been done to tackle the scheduling problem of in-memory computing-based frameworks like Apache Spark [71, 21, 50, 72]. However, most of these works assume the cluster setup to be homogeneous (there is only one type of VM instance for all the worker nodes) thus fail to make the scheduling technique cost-efficient from a cloud perspective.

As a motivating example, consider a cluster having 2 homogeneous VMs each having 8 CPU cores capacity. If a Spark job has 2 executors requirement with 2 cores for each, the total CPU cores requirement is 4. However, most of the existing strategies will use both the VMs to place these 2 executors which will lead to resource wastage and a higher VM usage cost. On the contrary, if a scheduler can consider the VM pricing model and different VM instance types in the cluster, executors from the jobs could be tightly packed in fewer cost-effective VMs. Thus, the instances with more resource capacity and higher price will be used only if there is a high load on the cluster. Therefore, in this chapter, we formulate the scheduling problem of Spark jobs in a cloud-deployed cluster as a variant of the bin-packing problem. Here, our primary target is to reduce the cost of VM usage while maximizing resource utilization and improving job performance.

In summary, our work makes the following key **contributions**:

- We propose two job scheduling algorithms. The first algorithm is a greedy algorithm adapted from the Best-Fit-Decreasing (BFD) heuristic, and the second algorithm is based on Integer Linear Programming (ILP). Both of these algorithms can improve cost-efficiency of a cloud deployed Apache Spark cluster. Besides, our proposed algorithms also prioritize jobs based on their deadlines and enhance job performance for network-bound jobs.

- We develop a scheduling framework by utilising Apache Mesos [10] cluster manager and this framework can be used to implement scheduling policies for any

Mesos supported data processing frameworks in addition to Spark.

- We implement the proposed algorithms on top of the developed scheduling framework.
- We perform extensive experiments with real applications and workload traces under different scenarios to demonstrate the superiority of our proposed algorithms over the existing techniques.

The rest of the chapter is organized as follows. In section 4.2, we discuss the background of Apache Spark and Apache Mesos. In section 4.3, we describe the existing works related to this chapter. In section 4.4, we show the motivating examples and formulate the scheduling problem. In section 4.5, we demonstrate the implemented prototype system. In section 4.6, we evaluate the performance of our proposed algorithms, show the sensitivity analysis of various system parameters and discuss the feasibility of our proposed algorithms. Section 4.7 concludes the chapter and highlights future work.

4.2 Background

We use Apache Spark as the target big data processing framework and Apache Mesos as the cluster manager where we implement our scheduling policies. In this section, we briefly introduce the basic concepts, system architecture, resource provisioning and scheduling mechanisms in these two frameworks.

4.2.1 Apache Spark

Apache Spark is one of the most prominent in-memory big data processing frameworks. It is a multi-purpose open-source platform with high scalability. Spark supports applications to be built with various programming languages like Java, Scala, R, Python etc. Besides, extensive and interactive analysis can be done using the available high-level APIs. Furthermore, a variety of input data sources like HDFS [11], HBase [14], Cassandra [15] etc. are supported by Spark. It outperforms traditional Hadoop-MapReduce based platform by conducting most of the computations in memory. In addition, results

from the intermediate stages are cached in memory for faster data re-processing. Spark uses *Resilient Distributed Dataset (RDD)* [16] for data abstraction which is fault tolerant by nature. In contrast to HDFS, Spark does not implement replication. Spark keeps track of how a specific piece of data was calculated, so it can recalculate any lost RDD partitions if a node fails or is shutdown by a scheduler. A Spark cluster follows a Master-Worker model, where there should be at least one *Master* node and one or more *Worker* nodes. However, multiple master nodes can be used by leveraging ZooKeeper [73]. From a cloud perspective, each master/worker node can be deployed in a cloud *VM*. Spark has its default standalone cluster manager which is sufficient to deploy a production-grade cluster. Moreover, it also supports popular cluster managers like Hadoop Yarn [12], Apache Mesos [10] etc.

When a Spark job/application is launched in a cluster, the *Driver* program of that job creates one or more executors in the worker nodes. *Executor* is a process of an application that holds a fixed chunk of resources (CPU cores, memory, and disk) and all the executors from the same job have identical resource requirements. Tasks are run in parallel in multiple threads inside each executor which lives during the entire duration of that job. As all the jobs have an independent set of executors, jobs are isolated, and each job's driver program can create its own set of executors and schedule tasks in them.

Resource allocation in a Spark cluster can be done in three ways: (1) Default: the user does not set any limits on the required resources for a job, and it uses all the resources of the entire cluster. Therefore, only one job can run in the cluster at a time and even if that job only requires a small chunk of resources, all the resources are allocated to it; (2) Static: if a user sets a limit on the required resources for a job, only that amount of resources will be allocated for that job, and any remaining resources can be assigned to any future job. Therefore, in this mode, it is possible to run multiple applications in the cluster and (3) Dynamic: resources are allocated similarly as the static allocation mechanism, but if any resource (CPU core only) is not utilized, it could be released to the cluster so that any other application can use it. Besides, this resource can be taken back from the cluster in future if needed by the original job.

By default, Spark supports FIFO scheduling across jobs. Therefore, jobs wait in a FIFO queue and run one after another. A new job is scheduled whenever any resources

are available to create any executor for the next job. Besides, Spark also has a FAIR scheduler, which was modelled after the Hadoop Fair Scheduler. Here, jobs can be grouped into pools, and different scheduling options can be set for each pool. For example, weight determines the priority of a job pool. By default, each pool has a weight 1, but if any pool is assigned 2 as the weight, it will get twice the resources than other pools. Within each job pool, jobs are scheduled in a FIFO fashion. Each pool also has a minimum share (minShare) of resources in the cluster, and a cluster manager only assigns more resources to a highly weighted pool once all the pools have met their minimum share of resources. By default, Spark spreads the executors from the same job into multiple workers for load balancing. In addition, the standalone cluster manager can also consolidate executors into fewer worker nodes (by greedily using the current worker node to place as many executors as possible). However, Spark assumes that all the worker nodes are homogeneous (same resource capacity), and it also does not consider the price of using a worker node (if it is deployed on cloud VM).

4.2.2 Apache Mesos

Apache Mesos is considered to be a data-center level cluster manager due to its capability of efficient resource isolation and sharing across distributed applications. It resides between the application and the OS layer and makes it easier to deploy and manage large-scale clusters. In Mesos, jobs/applications are called frameworks and multiple applications from different data processing frameworks like Spark, Strom, and Hadoop can run in parallel in the cluster. Therefore, Mesos can be used to share a pool of heterogeneous nodes among multiple frameworks efficiently. Mesos utilizes modern kernel features by using *cgroups* in Linux and *zones* in Solaris to provide isolation of CPU, memory, file system etc.

Mesos introduces a novel two-level scheduling paradigm where it decides a possible resource provisioning scheme according to the weight, quota or role of a framework and offers resources to it. The framework's scheduler is responsible for either rejecting or accepting those resources offered by Mesos according to its scheduling policies. If

https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/FairScheduler.html

a framework's scheduler accepts a resource offer from Mesos, the resources specified by that offer can be used to launch any computing tasks. Mesos also provides flexible Scheduler HTTP APIs which can be used to write custom user-defined scheduling policies on top of any big data processing platform. Besides, it provides Operator HTTP APIs to control the resource provisioning and scheduling of the whole cluster. Mesos supports dynamic resource reservation; thus resources can be dynamically reserved in a set of nodes by using the APIs and then a job/framework can be scheduled only on those resources. When a job is completed, resources can be taken back and reserved for any future job. It is a significant feature of Mesos as any external scheduler implemented on top of Mesos can have robust control over the cluster resources. Furthermore, the external scheduler can perform fine-grained resource allocation for a job in any set of nodes with any resource requirement settings. Lastly, various policies can be incorporated into an external scheduler without modifying the targeted big data processing platform or Mesos itself; so the scheduler can be extended to work with other big data processing platforms. For the benefits mentioned above, we have built a scheduling framework on top of Mesos to implement our proposed scheduling algorithms.

4.3 Related Work

Most of the data processing frameworks like Hadoop, Spark schedule jobs in a FIFO manner and distributes the tasks/executors from each job in a distributed round-robin fashion. To avoid resource contention FAIR scheduler was introduced for fair distribution of cluster resources among the jobs. In Mesos, scheduling is done by the Dominant Resource Fairness (DRF) [17] scheduling algorithm, which identifies the dominant resource type (CPU/memory) of each job. Then it offers resources to each job in such a way that overall use of cluster resources is well-balanced.

There has been a significant amount of research in the area of cluster scheduling. However, most of these schedulers focused Hadoop-MapReduce based clusters. Kc et al. [54] addressed the problems of Hadoop FIFO scheduler by introducing a dead-

http://mesos.apache.org/documentation/latest/scheduler-http-api/
http://mesos.apache.org/documentation/latest/operator-http-api/

Table 4.1: Related Work

Features	Related Work					Our Work
	DRF [17]	Quasar [71]	Morpheus [50]	Justice [72]	OptEx [21]	
Frameworks	✗	✗	✗	✗	✗	✓
VM types	✗	✗	✗	✗	✗	✓
Job types	✓	✓	✓	✓	✓	✓
Cost-efficient	✗	✗	✗	✓	✓	✓
Performance	✓	✓	✓	✓	✓	✓
Self-adaptive	✗	✓	✓	✓	✗	✓
Deadline	✗	✓	✓	✓	✓	✓

line constraint scheduler that prioritizes map/reduce tasks from each job based on their deadline. LATE [68] is a delay scheduler that targets to improve job throughput and response times by considering data locality into the scheduler in a multi-user MapReduce cluster. However, it treats the cluster setup to be homogeneous thus performs poorly in heterogeneous environments. SAMR [69] proposed a self-adaptive scheduling algorithm that classifies the performance of jobs from the historical data. It also identifies slow nodes dynamically and creates backup tasks so that MapReduce jobs will have a better performance in a heterogeneous environment. Tian et al. [70] considered job heterogeneity and proposed a triple-queue scheduler to keep the CPU and I/O bound applications isolated to improve the overall cluster performance. However, all of these works are focused on Hadoop-MapReduce performance modelling and scheduling and cannot be applied to an in-memory data processing framework like Spark.

As a platform like Spark has many configuration parameters, it is hard to set the appropriate resource requirement for a job. Wang et al. [61] tried to fine-tune Spark configuration parameters to improve the overall system performance. Gounaris et al. [62] investigated the problem of resource wastage that happens when a Spark application consumes all the nodes in a cluster. Gibilisco et al. [63] built multiple polynomial regression models on the application profile data and selects the best model to predict application execution time with unknown input data or cluster configuration. Wang et al. [19] tried to model application performance in DAG-based in-memory analytics platforms. Here, the execution times from multiple stages of a job are collected and then

used to predict the execution time. Islam et al. [20] focused on fine-grained resource allocation for Spark jobs with deadline guarantee. However, these works can only be applied to predict job-specific resource demands under homogeneous cluster environments.

There are a very few cluster schedulers [74] that support Spark jobs focusing on performance improvement and cost saving. Quasar [71] is a cluster management system that minimizes resource utilisation of a cluster while meeting user-provided application performance goals. It uses efficient classification techniques to find the impacts of resources on an application's performance. Then it uses this information for resource allocation and scheduling. It also dynamically adjusts resources for each application by monitoring resource usage. Morpheus [50] estimates job performance from historical data using performance graphs. Then it performs a packed placement of containers where it places a job that results in the minimal cluster resource usage cost. Moreover, Morpheus dynamically re-provisions failed jobs to improve overall cluster performance. Justice [72] is a fair share resource allocator that uses deadline information of each job and historical job execution logs in an admission control. It automatically adapts to workload changes and provides sufficient resources to each job so that it meets deadlines just in time. OptEx [21] models the performance of Spark jobs from application profiles. Then the performance model is used to schedule a cost-efficient cluster by deploying each job as a service in the minimal set of nodes required to satisfy its deadline.

The problems with most of the cluster schedulers are that they do not consider executor-level job placement. All of them only select the total number of resources or nodes needed for each job while making any scheduling decision. However, our scheduler takes advantage of VM heterogeneity (different types of VM instances) and uses smaller VMs for executor placement to minimize the overall resource usage cost of the whole cluster. Besides, most of the cluster schedulers use the round-robin placement of executors in the VMs while we consolidate the executors to use less number of VMs. Therefore, it minimizes inter-node communications for network-bound jobs thus improves the performance. A comparison of our approach with the existing works is illustrated in Table 4.1. It can be observed that our proposed solution considers multiple VM types in the scheduling algorithm. Moreover, we also provide a scheduling framework

to incorporate new scheduling policies.

Currently, commercial cloud service providers such as AWS and Windows Azure provide clusters and big data analytics services on the Cloud. For example, Apache Spark on Amazon EMR and Azure HDInsight. Besides job scheduling, there are many other ways to reduce costs in a commercial cloud computing platform. For example, EC2 spot instances and reserved instances have many features. Commercial cloud service providers optimize instance usage costs from their side by turning off idle instances. Our proposed approach complements these solutions by tight packing of executors in fewer instances so that those instances can be turned off. Hence, even if all the nodes are Spot instances, our approach is still cost-efficient as we use minimal number of instances as compared to the default Spark scheduler. While the commercial cloud service providers work on the VM instance level, our approach works on the executor level scheduling which is more fine-grained. Therefore, for the most cost-benefit, job scheduling from user-side also plays a vital role and while used in conjunction with commercial cloud providers' instance features, significant performance improvement and cost reduction can be achieved. Lastly, our approach can also be used for a local cluster which is deployed with on-premise physical resources.

4.4 Cost-efficient Job Scheduling

In this section, we explain the motivations of this work, the problem formulation, the proposed job scheduler and the executor placement algorithms and the complexity of the proposed algorithms.

4.4.1 Motivation

The utilization of resources in a big data cluster varies at different times of the day. For example, Fig. 4.1 depicts the job submission frequencies at different hours in a particular day from a Facebook Hadoop workload trace. There are several hours in a day when

https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark.html
https://azure.microsoft.com/en-au/services/hdinsight/
https://aws.amazon.com/emr/features/
https://github.com/SWIMProjectUCB/SWIM/wiki/Workloads-repository

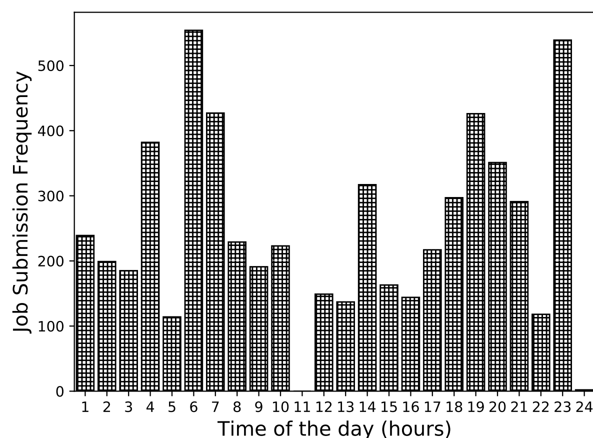


Figure 4.1: Job submission frequencies in a single day (Facebook Hadoop Workload Trace-2009)

the job submission rate is lower than usual. Therefore, if a big data processing cluster is deployed in the public cloud, it would be costly to keep all the VMs turned on as the cluster might not be fully utilised. However, the bill of using a VM is charged as pay-per-use basis and most of the cloud providers per-second billing period. Hence, if a VM is not used to schedule any jobs, it can be turned off to reduce the monetary cost of the cluster. The turned off VMs can be turned on again in future depending on the overall resource demands in the cluster.

Cloud service providers offer different types of VMs which have different pricing model. In general a small VM with lower resource capacity is cheaper than a large VM with high resource capacity. Therefore, if a cluster is deployed with different types of VM instances, smaller VMs can be used in the low-load period of the cluster to save cost whereas the bigger VMs can be utilized only in the high-load period.

Most of the cluster schedulers place the executors from each job in a distributed (round-robin) fashion in the VMs which has the following problems:

- VMs are under-utilized, and resources are wasted in all the VMs. This problem leads to a higher cost of using the whole cluster as most of the VMs are turned on at all times.

https://aws.amazon.com/blogs/aws/new-per-second-billing-for-ec2-instances-and-ebs-volumes
https://aws.amazon.com/ec2/pricing/on-demand/

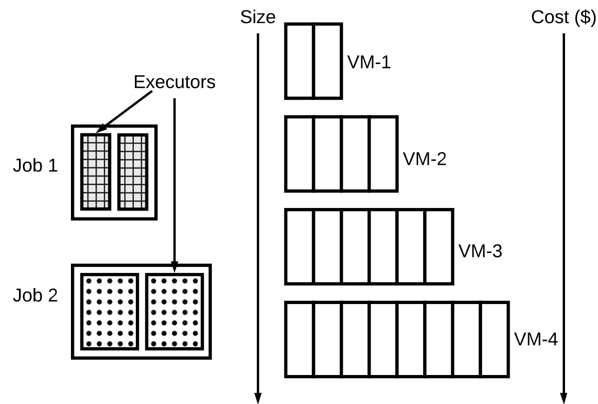
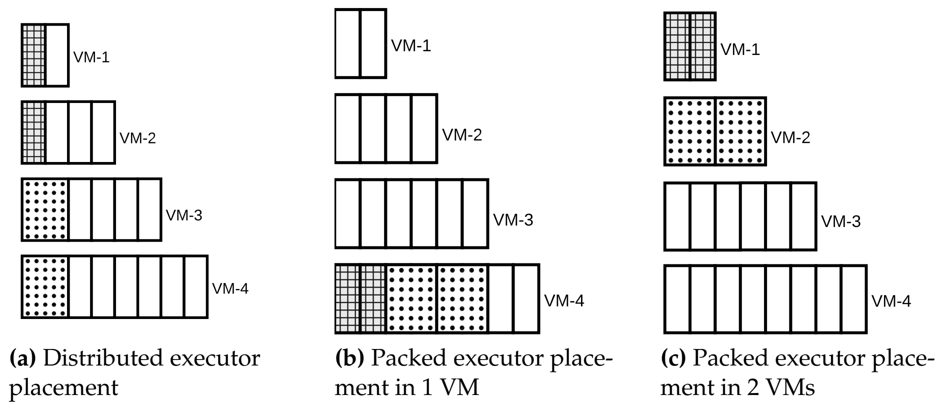


Figure 4.2: An example cluster with different types of Jobs and VMs



(a) Distributed executor placement

(b) Packed executor placement in 1 VM

(c) Packed executor placement in 2 VMs

Figure 4.3: Different Executor Placement Strategies

- In the cloud, different types of VM instances are available to use as the worker nodes and using only a single type of VM to compose a cluster might not be cost-effective. For example, a cluster has only one type of VM (16 CPU core, 64GB memory). If at a light-load hour only a single job is submitted (2 CPU core, 2GB memory), even using one VM would be costly. Using only small VM instances to compose a cluster would also fail as executors from different Spark jobs might have different size (resource requirement), so executors with high resource requirement will not fit in smaller VMs.

- For network-bound jobs, performance is reduced due to increased network transfers among the executors due to the distributed placement of executors in different VMs.

The consolidated executor placement option of Spark can not save cost as it does not consider the prices of different workers (VMs), and may choose the biggest VM to consolidate executors. Fig. 4.2 shows an example scheduling scenario where two jobs (with different resource demand) are submitted to a cluster composed of four VMs (with different resource capacity). For simplicity, let us assume that the executors from all the jobs require only one type of resource (e.g., CPU cores). The total number of slots in each VM represents its resource capacity. Similarly, the width of each executor of a job represents its resource demand. Therefore, in our example, each executor from job-1 requires 1 CPU core, and each executor from job-2 requires 2 CPU cores. VM-1, VM-2, VM-3, and VM-4 have a resource capacity of 2, 4, 6 and 8 CPU cores, respectively. In addition, the cost of using each VM is equivalent to its size, hence VM-1 is the cheapest VM whereas VM-4 is the costliest VM. Fig. 4.3a-4.3c depicts some of the possible executor placement strategies. Fig. 4.3a shows a distributed executor placement strategy (round-robin) which is used by most of the scheduling policies. In this placement, all the VMs are used but under-utilized. Therefore, this placement will lead to the highest VM usage cost. An alternative strategy which can be used in Spark to consolidate executors can be seen in Fig. 4.3b. However, as the cluster manager is unaware of the VM instance pricing or resource capacity, if it chooses to place job-1 in VM-4, job-2 will also be placed in VM-4 to consolidate executors from both jobs in fewer VMs. Even though Spark's

executor consolidation strategy provides a better VM usage cost than the round-robin strategy, it can be further improved as shown in Fig. 4.3c. Here, when job-1 first arrives it is placed in the cheapest VM (VM-1) where the executors of the current job fits properly. Then, job-2 is placed into the 2nd cheapest VM (VM-2), as VM-1 is already used. This strategy provides the cheapest VM cost usage even though executors are consolidated in more than one VM.

4.4.2 Problem Formulation

In an Apache Spark cluster, the resource requirements of the executors from the job are same. In addition, each worker node (VM) has a set of available resources (e.g., CPU cores, memory) which can be used to place executors from any job if the resource requirements are met. Therefore, for each submitted job in the cluster, the main problem is to find the placement of all its executors to one or more available VMs. Besides, resource capacity in each VM must not be exceeded while placing one or more executors in that VM during the scheduling process. As the compact assignment of executors leads to cost reduction due to fewer VM usages, we model the scheduling problem as a variant of the bin-packing problem. Table 4.2 shows the notations we use to formulate the problem.

We consider the resource requirement of an executor in two dimensions – CPU cores and memory. Therefore, each executor of a job can be treated as an item with multi-dimensional volumes that needs to be placed to a particular VM (bin) in the scheduling process. Suppose, we are given a job with E executors where each executor has CPU and memory requirements of τ_i^{cpu} and τ_i^{mem}, respectively ($i \in \xi$). There are K types of VM available each with a two-dimensional resource capacity (CPU, Mem) and incurs a fixed cost P_k, if used. The problem is to select VMs and place all the executors into these VMs such that the total cost is minimized and the resource constraints are met.

Table 4.2: Definition of Symbols

Symbol	Definition
job	The current job to be scheduled
E	Total executors required for job
ξ	The index set of all the executors of job, $\xi = \{1, 2, 3, ..., E\}$
Ψ	The index set of all the VM types, $\Psi = 1, 2, ..., K$
m_k	An upper-bound on the number of type k VMs
δ_k	The index set for each type k VM; $\delta_k = \{1, 2, ..., m_k\}, k \in \Psi$
P_k	Price of using a VM of type k
ω_{jk}^{cpu}	Available CPU in the jth VM of type k, $j \in \delta_k, k \in \Psi$
ω_{jk}^{mem}	Available Memory in the jth VM of type k, $j \in \delta_k, k \in \Psi$
τ^{cpu}	CPU demand of any executor of job
τ^{mem}	Memory demand of any executor of job
RA_{jk}	Resource Availability metric of the jth VM of type k
RD_{job}	Resource Demand metric for job

The optimization problem is:

$$\text{Minimize: } Cost = \sum_{k \in \Psi} P_k \left(\sum_{j \in \delta_k} y_{jk} \right) \tag{4.1}$$

$$\sum_{k \in \Psi} \sum_{j \in \delta_k} x_{ijk} = 1 \qquad \forall i \in \xi \tag{4.2}$$

$$\sum_{i \in \xi} (x_{ijk} * \tau^{cpu}) \leq \omega_{jk}^{cpu} * y_{jk} \qquad \forall k \in \Psi, j \in \delta_k \tag{4.3}$$

$$\sum_{i \in \xi} (x_{ijk} * \tau^{mem}) \leq \omega_{jk}^{mem} * y_{jk} \qquad \forall k \in \Psi, j \in \delta_k \tag{4.4}$$

$$x_{ijk},\ y_{jk} \in \{0, 1\}, \qquad \forall i \in \xi, k \in \Psi, j \in \delta_k$$

Cost Minimization: As shown in Eq. 4.1, our objective is to minimize the cost of using the whole cluster while scheduling any job. The total cost is modelled as the aggregated cost of using all the VMs. The binary decision variable y_{jk} is used which

controls whether VM j of type k is used or not.

$$y_{jk} = \begin{cases} 1 & \text{if the } j\text{th VM of type } k \text{ is used;} \\ 0 & \text{otherwise.} \end{cases}$$

Executor Placement Constraint: An executor can be placed only in one of the VMs and this placement constraint is denoted in Eq. 4.2. The binary decision variable x_{ijk} is used which controls whether executor i is placed on VM j of type k.

$$x_{ijk} = \begin{cases} 1 & \text{if executor } i \text{ is placed in } j\text{th VM of type } k\text{;} \\ 0 & \text{otherwise.} \end{cases}$$

Resource Capacity Constraints: The total resource demands of all the executors placed in a VM should not exceed the total resource capacity of that VM. The resource constraints for CPU cores and memory are shown in Eq. 4.3 and 4.4, respectively.

Bin packing is a combinatorial optimization problem and has proved to be NP-Hard [75]. The above optimization problem is an Integer Linear Programming (ILP) formulation of the multi-dimensional bin packing problem. When the scheduler has to schedule a job, the ILP model can be constructed by using the current job's resource demand and cluster resource availability. Then, it can be solved by exact methods such as Simplex [76], Branch and Bound [77] to find the most cost-effective executor placement for that job. However, constructing the ILP dynamically before scheduling each job can be time-consuming. Especially, if the problem size goes bigger (large cluster, or jobs with many executors), the ILP might not be feasible as it requires exponential time to solve. In this case, efficient heuristic methods can be used for faster executor placement.

4.4.3 Job Scheduler

The proposed job scheduler exhibits the following characteristics:

- The scheduler is online, that means it has no prior knowledge of job arrival and dynamically schedules jobs upon arrival

- The scheduler prioritizes jobs based on their deadline

- The scheduler tries to minimize the cost of VM usage while placing the executors of a job

Before discussing the scheduling algorithm, we introduce the important concepts used to design the scheduler.

Resource Unification Thresholds (RUT): As we have two types of resources (e.g., CPU and memory), the resource capacity of a VM and resource demand of a job cannot be represented with only one type of resource. Therefore, to holistically unify multiple types of resources, we introduce RUT and use it as a system parameter. Each of the thresholds acts as a weight for a single resource type, and the summation of these threshold values is 1 (Eq. 4.5). In our case, α is the threshold associated with CPU and β is the threshold associated with memory. Note that, this is a generalized unification which can be extended to multiple resource types depending on the system needs. A detailed discussion on how to assign Resource Unification Threshold (RUT) values is provided in section 4.6.6.

Resource Availability (RA_{jk}): It is a metric that represents the resource availability of a VM in the unified form. Eq. 4.6 and Eq. 4.7 shows the formula to compute the total amount of CPU and memory in the cluster, respectively. We use the formula shown in Eq. 4.8 to calculate RA_{jk} of a VM. Here, the currently available amount from each resource type is converted to the percentage of resource w.r.t the total cluster resource (of the same type) and then multiplied to the corresponding RUT. Then, the total resource capacity is found by summing these values.

$$\alpha + \beta = 1 \tag{4.5}$$

$$CPU_{total} = \sum_{k \in \Psi} \sum_{j \in \delta_k} \omega_{jk}^{cpu} \tag{4.6}$$

$$MEM_{total} = \sum_{k \in \Psi} \sum_{j \in \delta_k} \omega_{jk}^{mem} \tag{4.7}$$

$$RA_{jk} = \frac{\omega_{jk}^{cpu}}{CPU_{total}} * \alpha + \frac{\omega_{jk}^{mem}}{MEM_{total}} * \beta \tag{4.8}$$

$$RD_{job} = \left(\frac{\tau^{cpu}}{CPU_{total}} * \alpha + \frac{\tau^{mem}}{MEM_{total}} * \beta \right) * E \tag{4.9}$$

Resource Demand (RD_{job}): It is a metric that represents the resource demand of a *job* in the unified form. We first find the resource demand of one executor, then multiply it to the total executors to find the RD_{job} as shown in Eq. 4.9.

JobBuffer, JobQueue and DeadlineJobQueue: We use a $JobBuffer$ to hold all the incoming jobs that are submitted to the scheduler. Moreover, two priority queues: $JobQueue$ and $DeadlineJobQueue$ are used to keep regular and deadline-constrained jobs, respectively. In $JobQueue$, jobs are kept sorted in descending order of their resource demand (RD_{job}). Jobs are kept sorted based on the Earliest Deadline First (EDF) strategy in the $DeadlineJobQueue$. The scheduler can transfer jobs from the $JobBuffer$ to the priority queues at any time.

Algorithm 3 shows the policy used by the proposed scheduler. When the scheduler starts, at first it fetches deadline-constrained jobs from the $JobBuffer$ (line 3). As $DeadlineJobQueue$ is kept sorted based on EDF, if a newly added deadline-constrained job has a tighter deadline than the already awaiting jobs, it will be extracted from the queue to be scheduled before any other jobs (line 7). If the $PlaceExecutor()$ procedure returns success in finding VMs to place the executors, the job will be launched in the cluster (lines 8-9). The scheduler is not preemptive, so when a job is scheduled (whether it is a regular or a deadline-constrained job), it will not be killed or suspended. Therefore, while any deadline-constrained jobs are waiting and the cluster does not have sufficient resources to execute them ($PlaceExecutor()$ procedure returns failure), the scheduler does not fetch any regular jobs until all the deadline-constrained jobs are scheduled.

Algorithm 3: Algorithm for the Job Scheduler

Input: $JobBuffer, JobQueue, DeadlineJobQueue$

1 **while** *SchedulerTerminationSignal* $\neq$ *true* **do**
2   **while** *true* **do**
3         $FetchDeadlineJobs(JobBuffer)$
4   **if** $DeadlineJobQueue = \phi$ **then**
5   **break**
6   **end**
7         $Job = ExtractJob(DeadlineJobQueue)$
8   **if** $PlaceExecutor(Job)$ *is successful* **then**
9             $LaunchJob(Job, PlacementList)$
10   **end**
11   **end**
12   **while** *true* **do**
13         $FetchRegularJobs(JobBuffer)$
14   **if** $DeadlineJobQueue \neq \phi$ **then**
15   **break**
16   **end**
17         $Job = ExtractJob(JobQueue)$
18   **if** $PlaceExecutor(Job)$ *is successful* **then**
19             $LaunchJob(Job, PlacementList)$
20   **end**
21   **end**
22 **end**

If there are no deadline-constrained jobs to schedule (line 4), only then the scheduler fetches regular jobs (line 13). Otherwise, it keeps trying to place executors for deadline-constrained jobs.

Before scheduling any regular jobs, the scheduler always checks whether any new deadline-constrained job has arrived. If so, it goes back to schedule those jobs (line 14-15). Otherwise, it starts scheduling regular jobs (lines 17-19). In some cases, it might be difficult to place a regular job with huge resource demand (as the $JobQueue$ is kept sorted in decreasing order of resource demand for jobs). In these cases, the scheduler skips the current job and tries to schedule the next job from the $JobQueue$.

4.4.4 Executor Placement

We propose two algorithms for cost-effective executor placements for any job in the cluster. The first algorithm constructs the Integer Linear Programming (ILP) model as shown in section 4.4.2 and tries to solve the ILP problem to find the most cost-effective executor placement for the current job. The second algorithm uses a greedy approach which is a modified version of the Best Fit Decreasing (BFD) heuristic to solve bin packing problems. Both of these algorithms can be used as the $PlaceExecutor()$ procedure of Algorithm 3.

ILP-based Executor Placement:

Algorithm 4 shows the ILP-based executor placement approach. At first, the cluster status is updated to obtain the latest resource availability of each VM. After this step, the optimization target, executor placement constraints, and resource capacity constraints are dynamically generated by using the current cluster resource availability and the resource demand for the executors of the current job. Then the constructed ILP problem is solved (by an ILP solver). If a feasible solution is found, the $PlacementList$ is returned which contains the chosen VMs where the executors can be created. Otherwise, if the modelled problem is not solvable, a failure is returned. Note that, when the constraints of resource availability are generated before scheduling each job, the VMs which are already used by other jobs will be set ($y_{jk} = 1$) so that the cost of using that machine will

be taken into account in the optimization target. Therefore, if there are any free resources available in the used VMs, the ILP solver will automatically try to fit as many executors as possible in those VMs before using any new VM to optimize cost.

Algorithm 4: ILP-based Executor Placement Algorithm

```
   Input: Job, the current job to be scheduled
   Output: PlacementList, a list of VMs where the executors of Job will be placed
 1 Procedure PlaceExecutor(Job)
 2     PlacementList ← ϕ
 3     Update Cluster Resource Availability
 4     Generate Optimization target (Eq. 4.1)
 5     Generate Executor Placement Constraints (Eq. 4.2)
 6     Generate Resource Capacity Constraints (Eq. 4.3,4.4)
 7     Solve ILP Problem
 8     if ILP is solved then
 9         return PlacementList
10     end
11     return Failure
12 end
```

BFD Heuristic-based Executor Placement:

To find the VMs where a job's executors can be placed, our proposed scheduler also uses a greedy algorithm. Algorithm 5 shows the procedure *PlaceExecutor*() which can be used to find the executor placement of any job. At first, the *VMList* (a list of used VMs in the cluster) is sorted based on an ascending order of Resource Availability (RA_{jk}) of the VMs (line 3). Then, it iterates all the VMs (line 4) and checks whether the current VM's resource availability satisfies an executor's resource demand (line 5). If so, it updates the resource availability of that VM (line 6) and adds this VM to a list called *PlacementList* (line 7). Instead of looking at the next VM, the current VM is greedily used to place as many executors as possible so that we have a tight packing of the executors and use a fewer number of VMs in the cluster. If this procedure finds placements for all the executors of a given job, it returns the *PlacementList* (lines 8-9). If the VMs in *VMList* are not sufficient to place all the executors, and the cluster has unused VM(s) (line 13), the smallest VM that satisfies the resource constraints will be turned on (line 14) and

Algorithm 5: BFD Heuristic-based Executor Placement Algorithm

```
   Input: Job, the current job to be scheduled
   Output: PlacementList, a list of VMs where the executors of Job will be placed
1  Procedure PlaceExecutor(Job)
2      PlacementList ← φ
3      Sort(VMList)
4      forall VM ∈ VMList do
5          while Placement of an executor in VM satisfies the constraints (Eq. 4.3,4.4) do
6              Update Resource Availability in VM
7              PlacementList.add(VM)
8              if PlacementList.size = E then
9                  return PlacementList
10             end
11         end
12     end
13     if Cluster has unused VM(s) then
14         Turn on the smallest VM_new that satisfies the constraints (Eq. 4.3,4.4)
15         VMList ← VMList ∪ VM_new
16         goto step 3
17     end
18     return Failure
19 end
```

added to the $VMList$ (line 15). Then the placement finding steps will be repeated (line 16). Otherwise, if the cluster does not have sufficient resources to place all the executors of the current job, a failure will be returned (line 18).

4.4.5 Complexity Analysis

To calculate the worst-case time complexity of Algorithm 3, we first assume that, p and r is the total number of deadline-constrained and regular jobs, respectively that need to be scheduled. If the total number of VM in the cluster is m, the time required to sort the $VMList$ is $mlog(m)$. If an exact algorithm is used to solve the ILP model built in Algorithm 4, the worst-case time complexity is $O(2^n)$ where n is the maximum number of slots available for placing executors across all the VMs. However, the worst-case time complexity of the BFD-based greedy approach shown in Algorithm 5 is $O(me)$, where e is the maximum number of possible executors for any job. There-

fore, if ILP-based executor placement is used, the worst-case time complexity of Algorithm 3 is, $O((2^n mlog(m))(p+r))$. Thus, it might require exponential time to complete the scheduling process for ILP based approach. In contrast, for the BFD-based executor placement, Algorithm 3 has a polynomial worst-case time complexity of $O((m^2 log(m))(p+r))$.

4.5 System Design and Implementation

We design a scheduler on top of the Mesos cluster manager instead of modifying the native Spark scheduler to implement our scheduling algorithms. The benefit of keeping a separate module for the scheduler without extending the existing framework is two-fold. First, it can be extended to work with any other data processing frameworks supported by Mesos. Second, it can be used as a generic scheduling framework so that new policies can be incorporated into the scheduler. The prototype scheduler can be treated as an external scheduler in the system architecture as depicted in Fig. 4.4. The implementation of the prototype system is open-source so that it can be used or extended by the research community.

The external scheduler can be installed in any VM, but in our case, we plugged it in the Mesos master node and ran it as a separate application alongside with the Mesos master process. Users submit jobs to the external scheduler and depending on the scheduling policy, the scheduler provisions resources in the cluster and launch any job with the help of Mesos master. In the architectural diagram shown in Fig. 4.4, dashed lines represent job submission or executor creation flow where solid lines represent the control flows of the scheduler. As discussed previously in the algorithm section, there are three data structures to keep the jobs in the scheduler: Job buffer (to hold the incoming jobs), deadline queue (to hold deadline-constrained jobs), and job queue (to hold regular jobs). When the scheduler decides to schedule a job in the cluster, at first, it uses the Mesos HTTP APIs and sends JSON formatted request messages to Mesos master HTTP API endpoints to dynamically reserve resources. After getting the acknowledgment of successful resource reservation by the Mesos master, it launches that job through

https://github.com/tawfiqul-islam/SLA-Scheduler

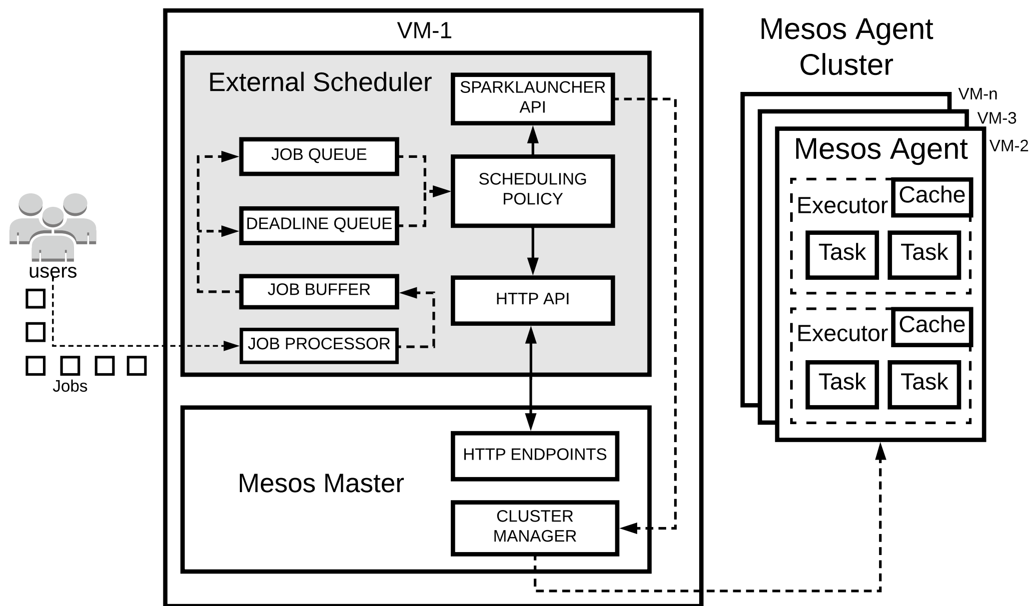


Figure 4.4: The implementation of the prototype system on top of Apache Mesos

the Mesos cluster manager by using the SparkLauncher APIs. At this stage, the driver program of the launched Spark job takes control and creates executor(s) in one or more VMs by using the reserved resources only. At any point of the scheduling process, if a VM is unused and no jobs are currently reserved on it for any future jobs to be scheduled, it is turned off by the scheduler to save resource usage cost. Additionally, the scheduler can also turn on one or more VMs if the currently available resources in the active VMs is not sufficient to schedule new jobs.

We have implemented this pluggable external scheduling framework in Java. We have used SCPSolver API with LPSolve Solver Pack library to solve the proposed ILP-based executor placement model in the scheduler. To implement the automatic VM turn on/off mechanism from the scheduling process, we have developed a module by using OpenStack Boto3 library. However, this module can be easily extended to support any other cloud service providers by using their APIs. The scheduler also uses Mesos scheduler HTTP API and operator HTTP API to control the resource provisioning in the cluster. The Mesos master accepts messages in JSON format while communicating through

http://scpsolver.org/
http://lpsolve.sourceforge.net/5.5/
https://boto3.readthedocs.io/en/latest/

the HTTP APIs. Therefore, java-json API was used to construct/parse JSON formatted messages. Furthermore, SparkLauncher API was used to automate Spark job submission from the scheduler. The scheduler accepts job submission requests from the users through a job processor interface that listens on a configurable TCP port. Job submission requests to the scheduler should be constructed in JSON format with some simple fields. In a job submission request, the users have to specify the details of a job having the following fields: job-id, input-path, output-path, application-path, application-main-class, resource requirement (CPU cores, memory in GB and total-executors) and an optional application argument (e.g., iteration).

4.6 Performance Evaluation

In this section, we first provide the experimental setup details which includes the cluster resource configurations, benchmark applications, and baseline schedulers. Then we show the evaluations of the proposed algorithms in terms of cost, job performance, deadline violations, and scheduling overhead. Moreover, we also provide a sensitivity analysis of the system parameters and discuss the applicability of the proposed algorithms.

4.6.1 Experimental Setup

Cluster Configuration:

We have used Nectar Cloud, a national cloud computing infrastructure for research in Australia to deploy a Mesos cluster. It is a cluster consisting of three different types of VM instances. The detailed VM configurations and quantity used from each type with their similar pricing in Amazon AWS (Sydney, Australia) is shown in Table 4.3. In summary, our experimental cluster has 14 VMs with a total CPU (cores) of 100 and memory of 400GB. In each VM, we have installed Apache Mesos (version 1.4.0) and Apache Spark

http://www.oracle.com/technetwork/articles/java/json-1973242.html

https://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/launcher/package-summary.html

https://nectar.org.au/research-cloud/

(version 2.3.1). One m1.large type VM instance was used as the Mesos master while all the remaining VMs were used as Mesos Agents. The external scheduler was plugged into the Mesos master node. Spark supports different input sources as mentioned before, and the users can select which data sources they want to use. However, HDFS is the most prominent distributed storage service as it is highly scalable, and provides fault-tolerance through replication. Generally, HDFS keeps replica of a storage block in 3 datanodes. Hence, if any of these datanodes (VMs) are turned-off to save cost, HDFS will automatically create replicas on the available VMs. However, a storage block might be lost if all the 3 datanodes where its replicas reside are turned off. Therefore, in this special case, the VM turn on/off module should be modified to allow HDFS to create replicas before shutting down all the datanodes. For the simplicity of the current system implementation to test our proposed approach, we have mounted a 1TB volume in the master node and created a *Network File System (NFS)* to share this storage space with all the Mesos agents. As the NFS server is running on the master node which will not be turned off, the current implementation does not need to consider about data loss due to VM turn off. In addition, the performance overhead due to fetching the input data from the NFS server is negligible as it is only done once at the beginning of the jobs execution, and all the intermediate results are stored in each VMs local storage which is managed by Spark. For providing fault-tolerance, we plan to extend our implementation to work with HDFS in the future. We have used Bash scripting to automate the cluster setup process so that a large-scale deployment can also be conducted through these scripts. Furthermore, an existing cluster can also be scaled up if more VMs are provisioned from the Cloud service provider.

Table 4.3: Experimental Cluster Details

Instance Type	CPU Cores	Memory (GB)	Pricing (AWS)	Quantity
m1.large	4	16	$0.24/h	6
m1.xlarge	8	32	$0.48/h	5
m2.xlarge	12	48	$0.72/h	3

Benchmarking Applications:

We have used BigDataBench [67], a big data benchmarking suite to evaluate the performance of our proposed algorithms. We have chosen three different types of applications from BigDataBench, namely WordCount (compute-intensive), Sort (memory-intensive) and PageRank (network/shuffle-intensive). Each application was used to generate a workload where each job in a workload has varying input size ranging from 1GB to 20GB (for WordCount and Sort) or iterations ranging from 5 to 15 (for PageRank). To generate a heterogeneous workload, we have randomly mixed the previously mentioned different types of applications. We have extracted the job arrival times from two different hours of a particular day from the Facebook Hadoop workload trace. From a high-load hour, 100 jobs are used, and from a light-load hour, 50 jobs are used. The arrival rate of jobs in the high-load hour is higher than the light-load hour. Therefore, in the high-load hour, most of the resources are overwhelmed with jobs while in the light-load hour, the cluster is slightly under-utilized. The job profiles are collected by first submitting each job to run independently (without any interference from other jobs) in the cluster. Then the job completion time is averaged from multiple runs (5 for each job). While generating a workload, each job's average completion time is used as a hard deadline.

Baseline Schedulers:

The problem with most of the cluster schedulers for Spark jobs is that they do not consider executor-level job placement. Most of these approaches only select the total number of resources or nodes (VMs) needed for each job while making any scheduling decisions. However, our approach works on a fine-grained level by incorporating executor placements in job scheduling. Therefore, the existing works can not be directly compared with our proposed approach. The following schedulers are compared with our proposed scheduling algorithms:

- FIFO: The default FIFO scheduler of Apache Spark deployed on top of Apache Mesos. It schedules jobs on a first come first serve basis. We have used the con-

https://github.com/SWIMProjectUCB/SWIM/wiki/Workloads-repository

solidation option of the scheduler so that it tries to pack executors in fewer VMs instead of distributing executors on a round-robin fashion. As most of the existing scheduling algorithms use this default approach for executor placement, and it is also the common choice of a user with Spark jobs, we chose this scheduler to be one of the baselines.

- Morpheus [50]: We have adapted the executor placement policy of Morpheus. In this policy, *lowcost* packing is used for executor placement. Depending on the current cluster load, this policy finds the scarce resource demand (e.g., memory or CPU cores) of each job (Eq. 4.10). Then jobs are sorted in increasing order of their scarce resource demands. Therefore, resources in the cluster are well-balanced throughout the scheduling process so that more jobs can be executed in the long run. As Morpheus also uses a packing based approach for executor placement, we chose it as a baseline.

$$c_{job} = Max\left(\frac{CPU_{load} + CPU_{job}}{CPU_{total}}, \frac{MEM_{load} + MEM_{job}}{MEM_{total}}\right) \quad (4.10)$$

Note that, Spark dynamic resource allocation feature was turned on for both the baseline and the proposed scheduling algorithms.

4.6.2 Evaluation of Cost Efficiency

In this evaluation, we show the applicability of our proposed scheduling algorithms to different types of applications while reducing the cost of using a big data cluster. To calculate the total cost incurred by a scheduler, we save the status of a VM (whether it was turned on or off) in each second. Lastly, all the per-second costs ($cost_i$, cost incurred in ith second, $i = 1, 2, 3, ...T$; T=total makespan of the scheduler) incurred by a scheduler is calculated by using Eq. 4.1. Then all these per-second costs are summed for the whole makespan of the scheduling process as shown in Eqn. 4.11 to find the $Total_{cost}$.

$$Total_{cost} = \sum_{i \in T} Cost_i \quad (4.11)$$

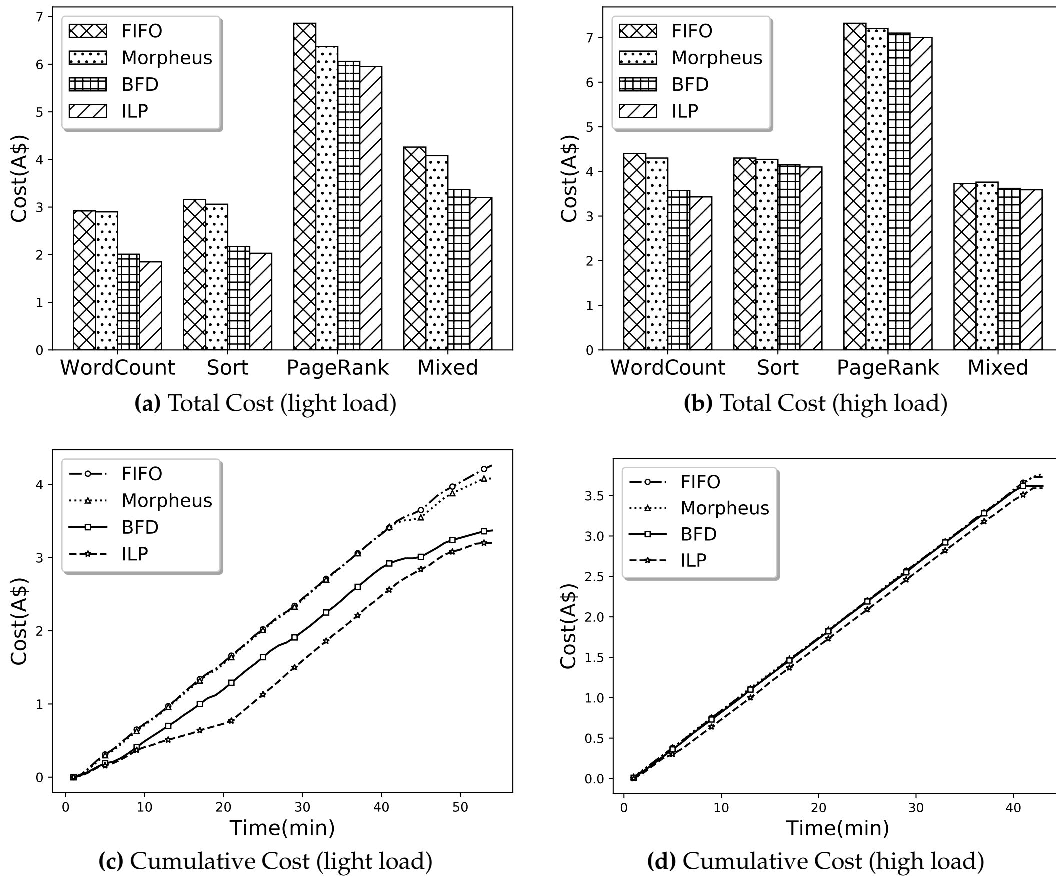
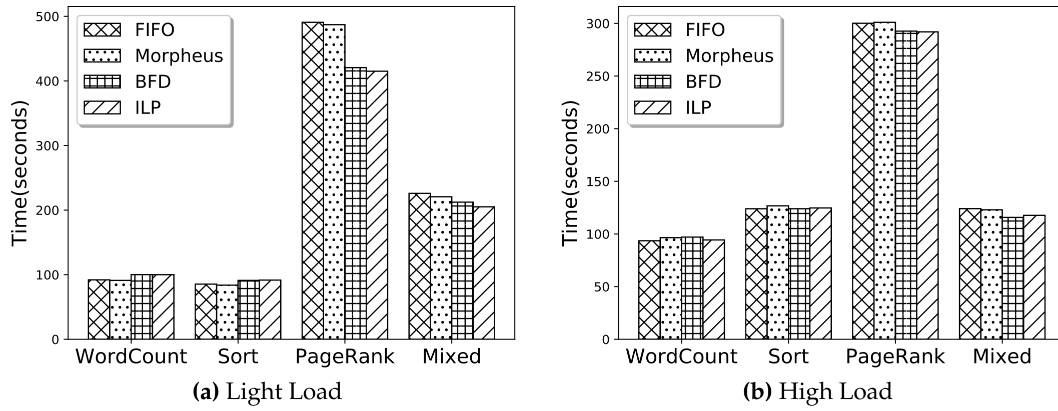


Figure 4.5: Cost comparison between the scheduling algorithms under different workload types

Fig. 4.5 depicts cost comparison between the scheduling algorithms under different workload types. The bar charts in Fig. 4.5a and Fig. 4.5b show the total cost incurred by different scheduling algorithms in the light-load and high-load hour, respectively. As our proposed scheduling algorithms use bin packing to consolidate the executors to a minimal set of VMs, the cost is reduced significantly as compared to other schedulers. In general, the ILP-based scheduling algorithms incur slightly lower cost than the BFD-based scheduling algorithm in all the scenarios as it can find the cost-effective executor placement for a job. Moreover, Morpheus performs slightly better than FIFO to lower the cost, because it prioritizes jobs in such a way that cluster resources are well-balanced to execute more jobs in the overall scheduling process.

As shown in Fig. 4.5a, both BFD-based and ILP-based scheduling algorithms exhibit significant cost reductions during the light-load hour. As compared to baseline scheduling algorithms, BFD and ILP reduce the cluster usage cost by at least 30% and 34%, respectively for WordCount and Sort applications. For PageRank application, BFD and ILP reduce the resource usage cost by at least 12% as compared to FIFO. Moreover, BFD and ILP reduce the resource usage cost by at least 5% as compared to Morpheus. As our proposed scheduling algorithms try to place the executors from the same job in fewer nodes (VMs), most of the shuffle operations happen intra-node thus improving job performance which results in overall cost reduction for network-bound applications. In the case of the mixed workload, BFD and ILP reduce the resource usage cost by 21% and 25%, respectively as compared to FIFO. Furthermore, BFD and ILP reduce the resource usage cost by 17% and 22%, respectively as compared to Morpheus. In the case of the high-load hour as shown in Fig. 4.5b, the cost reduction is smaller than the light-load period as the cluster is over-utilized. In this scenario, BFD and ILP show about 5-20% of cost reduction in different workloads.

Fig. 4.5c and Fig. 4.5d represents the cumulative VM cost by different scheduling algorithms during the whole scheduling process for the mixed workload in the light load and high load hours, respectively. It can be observed that in the high-load hour, the cumulative cost graph of all the scheduling algorithms look similar as it is not possible to reduce the cost significantly of an over-utilized cluster. However, in the light-load hour, the cost savings can be observed to increase over time for both BFD and ILP.



(a) Light Load **(b)** High Load

Figure 4.6: Comparison between the scheduling algorithms regarding average job completion times under different workload types

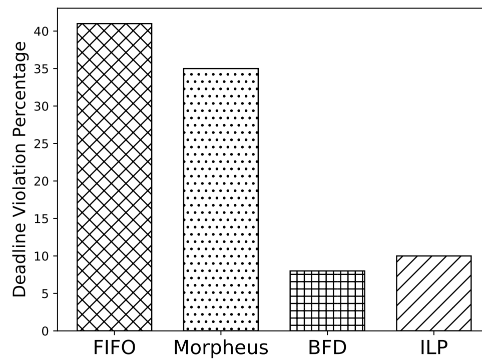


Figure 4.7: Comparison of deadline violations by different scheduling algorithms

4.6.3 Evaluation of Job Performance

Fig. 4.6a and 4.6b report the average job completion times for different scheduling algorithms in light-load and high-load hours, respectively. It can be observed that for WordCount and Sort applications, sometimes FIFO and Morpheus perform slightly better than our proposed algorithms. As our algorithms use fewer VMs to place all the executors, these VMs are stressed as both CPU cores, and memory resources are used at full capacity. However, it is negligible as compared to the total resource cost usage by the baseline schedulers. On the contrary, network-bound applications such as PageRank reduces the performance of both FIFO and Morpheus due to excessive network communications during the shuffle periods. Therefore, both BFD and ILP outperform the baseline algorithms in case of PageRank and mixed applications. As all the algorithms perform similarly for CPU/memory intensive applications, performance benefits in mixed workload mainly depend on the proportion of network-intensive applications. In the high-load hour, the cluster is overloaded with jobs so it might not be possible to consolidate the executors from the same job in fewer VMs. Therefore, the performance benefits can be observed to be higher in the light-load hour than the high-load hour for the mixed and PageRank applications. In the light-load hour, our proposed algorithms improve job completion time for at least 14% and 5% for PageRank and mixed applications, respectively. In the high-load hour, our algorithms improve job completion time for at least 3% and 5% for PageRank and mixed applications, respectively.

4.6.4 Evaluation of Deadline Violation

In this evaluation, we compare the percentage of deadline violations of different scheduling algorithms. This performance metric (θ^d) is found by using Eq. 4.12 where θ_m and θ_s is the number of missed and satisfied deadlines by a scheduler, respectively.

$$\theta^d = \frac{\theta_m}{\theta_s} \times 100\% \tag{4.12}$$

Both FIFO and Morpheus do not consider deadline-constrained jobs. In FIFO, a high priority job with the earliest deadline has to wait in the scheduling queue if it is sub-

mitted after one or more non-priority jobs. It will be scheduled only after executing all the previously arrived jobs. Morpheus determines the job priority by itself, where a job which results in the most balanced distribution of resources in the cluster (if that job is scheduled) will have the highest priority. However, in reality, top priority deadline-constrained jobs might not provide balanced resource distributions upon placement. Therefore, other non-priority jobs will be executed before these jobs. Both BFD and ILP use a simple Earliest Deadline First (EDF) strategy. Thus, all the jobs are kept sorted according to their deadlines, and the job with the earliest deadline is scheduled first. Fig. 4.7 depicts the deadline violation percentage of different schedulers. For this experiment, we have executed a heterogeneous mix (different application types) of priority (strict deadline) and non-priority jobs to measure the deadline violations by each scheduler. For FIFO and Morpheus, deadline violation occurred for 41% and 35% of jobs, respectively. However, both BFD and ILP were able to meet the deadlines for most of the jobs and have deadline violation percentage of only 8% and 12%, respectively. ILP has slightly higher deadline violation than the BFD because sometimes it takes a significant time to find the most cost-effective placement by this approach which causes deadline misses.

4.6.5 Evaluation of Scheduling Overhead

In this evaluation, we compare the scheduling delays caused by different scheduling algorithms. It is found by measuring the time it takes to find the executor placements of a job. Table 4.4 records the average scheduling delays by different scheduling algorithms under different workload types in both high-load and light-load hours. It can be observed that the native FIFO is the fastest among all the schedulers with scheduling delays averaging only from 2ms to 4ms. Both Morpheus and BFD are also fast as their average scheduling delay varies in the range from 3ms to 5ms and 4ms to 6ms, respectively. In contrast, as the ILP tries to find the most cost-effective executor placement for each job, in some cases it might require exponential time to complete. The results also indicate the same as the average scheduling delay varied from 0.65 seconds to up to 3.31 seconds for ILP. Although most of the jobs had a scheduling delay within 1 sec-

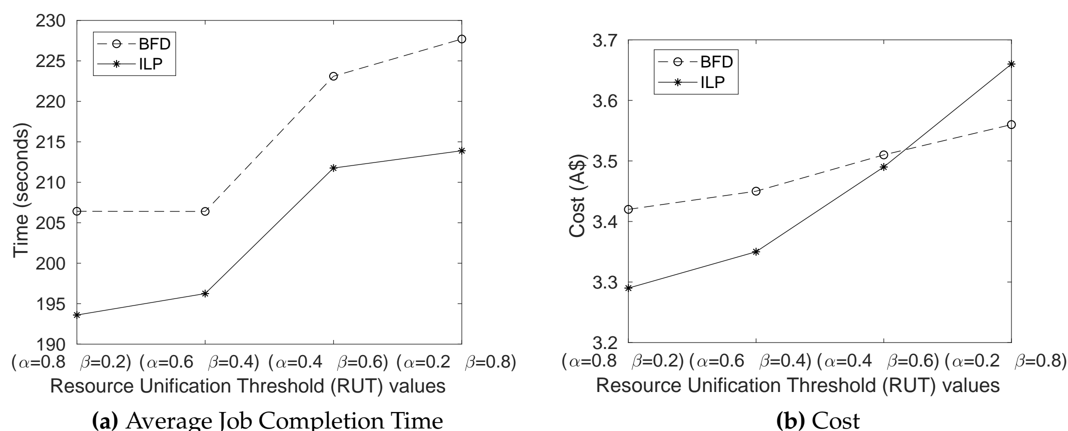
Table 4.4: Comparison of Average Scheduling Delays (unit: seconds) of different scheduling algorithms

Schedulers	Light-load				High-load			
	WC	**Sort**	**PR**	**Mixed**	**WC**	**Sort**	**PR**	**Mixed**
FIFO	0.002	0.004	0.002	0.004	0.003	0.003	0.003	0.004
Morpheus	0.004	0.004	0.003	0.005	0.005	0.004	0.003	0.004
BFD-based	0.006	0.005	0.005	0.004	0.005	0.004	0.004	0.005
ILP-based	3.31	3	0.75	1.92	0.73	2.63	0.65	1.3

ond, for the ILP, the average is higher as for some jobs it took about 3-4 minutes. The higher scheduling delay of ILP-based scheduling algorithm might cause some deadline misses. It can also be observed in Fig. 4.7 that, ILP-based scheduling algorithm has a slightly higher deadline miss percentage than the BFD-based algorithm. However, this performance degradation is negligible as compared to the baseline scheduling algorithms. Furthermore, for regular jobs or periodic jobs (e.g., long-running data analytics) that do not have strict deadlines, using the ILP-based scheduling algorithm is preferred as it can provide better cost reduction in the long run.

4.6.6 Effects of Resource Unification Thresholds (RUT)

RUT is a system parameter, and we have performed a sensitivity analysis to demonstrate the effects of it on both cluster usage cost and job performance. In our experimental cluster, we have two types of resources (e.g., CPU cores and memory). Resource unification thresholds (RUT) play a vital role in the scheduling process by acting as a weight while combining these two types of resources to determine the resource capacity of the VMs or the resource demand of the jobs. We have associated α as the RUT for CPU cores and β as the RUT for memory. The proper balance between RUT values depends on both the VM instance types and the workload types. Fig. 4.8 represents the effects of different RUT values on both average job completion time (Fig. 4.8a) and resource usage cost (Fig. 4.8b). This analysis was done by running both BFD and ILP-based scheduling al-



(a) Average Job Completion Time

(b) Cost

Figure 4.8: Effects of Resource Unification Threshold (RUT) values on average job completion time and cost

gorithms with the mixed workload. It can be observed from the figure that, decreasing the α value and increasing the β value tends to increase both average job completion time and resource usage cost in our experimental cluster. As using $\alpha = 0.8$ and $\beta = 0.2$ gives us both lower cost and job completion time, we use these RUT values in our experiments.

RUT values can also be tuned to give more priority to specific VMs or jobs. For example, if a cluster has more memory-bound jobs, to prefer VMs which have more memory to fit these jobs correctly, the β value can be increased, and α value can be decreased so that VMs which have high memory capacity/availability are preferred in the scheduling process. Similarly, jobs can also be prioritized based on their demand on a particular resource-type by adjusting the corresponding RUT values.

4.6.7 Discussion

The proposed scheduling algorithms can be applied to optimize the cost of using a cloud-deployed Apache Spark cluster. Our performance evaluation results show that the BFD heuristic-based approach performs very close to the ILP-based approach in all the cases. However, the ILP-based approach might have significant scheduling delays for a large cluster (many VMs). Therefore, in this case, we recommended using the BFD-based scheduling algorithm as it gives similar results with a small scheduling overhead

identical to the native FIFO. Another approach could be using both algorithms and using a time-constraint in the ILP. If the ILP can be solved within the time-constraint, the executor placements found by this approach will be used. Otherwise, the solution from the BFD-based approach will be used.

The proposed approach can also be used with HDFS. As HDFS generally creates replicas in 3 datanodes (VMs), if all these 3 VMs are selected to be turned off in the scheduling process to save cost, a storage block which was only saved in these 3 VMs will be lost. To mitigate this issue, it is not required to modify the scheduling algorithms. However, the VM turn on/off module should be modified for allowing HDFS to create replicas before shutting down a VM (datanode).

4.7 Summary

Scheduling is a challenging task in big data processing clusters deployed on the cloud. It gets even harder in the presence of different types of VMs and job heterogeneity. Most of the existing schedulers only target on improving job performance. In this chapter, we have used bin packing to formulate the scheduling problem and proposed two dynamic scheduling algorithms that enhance job performance and minimize resource usage cost. We have built a prototype system on top of Apache Mesos which can be extended to incorporate new scheduling policies. Therefore, this system can be used as a scheduling framework. We have demonstrated the outcomes of our extensive experiments on real datasets to prove the applicability of the proposed algorithms under various workload types.

Moreover, we have compared our algorithms with the existing baseline schedulers. The results suggest that our proposed scheduling algorithms reduce resource usage cost up to 34% in a cloud-deployed Apache Spark cluster. Furthermore, both network-bound and mixed jobs gain performance benefits (up to 14%) from tighter packing of executors in fewer VMs. We have also done the sensitivity analysis of the system parameter and discussed the effects of it on both cost and job performance. Lastly, we have discussed the feasibility of the proposed approach.

Chapter 5

Scheduling Big Data Applications in Hybrid Cloud

Due to the limited resource availability, the local or on-premise computing resources are often not sufficient to run big data jobs. Therefore, public cloud resources can be hired on a pay-per-use basis from the cloud service providers to deploy a Spark cluster entirely on the cloud. Nevertheless, using only cloud resources can be costly. Hence, now-a-days, both local and cloud resources are used together to deploy a hybrid cloud computing cluster. However, scheduling jobs in a cluster deployed on hybrid cloud is challenging in the presence of various Service-Level Agreement (SLA) demands such as cost minimization and job deadline guarantee. Most of the existing works either consider a public or a locally deployed cluster and mainly focus on improving job performance in the cluster. In this chapter, we propose efficient scheduling algorithms that leverage from different cost models in a hybrid cloud deployed cluster to optimize the Virtual Machine (VM) usage cost for both local and cloud resources and maximize the job deadline meet percentage.

5.1 Introduction

ANALYSING data at massive scale is becoming crucial due to the availability of huge data in various domains such as scientific research, social media, business. Several prominent big data processing platforms such as Hadoop [12], Spark [6], Storm [78] are used to analyze this enormous volume of data. A big data processing platform can be deployed in local-premises using computing resources owned by a company.

This chapter is derived from:

- **Muhammed Tawfiqul Islam**, Huaming Wu, Shanika Karunasekera, and Rajkumar Buyya, ”SLA-based Scheduling of Spark Jobs in Hybrid Cloud Computing Environments”, *IEEE Transactions on Computers (TC)* [Under 2nd Revision].

Besides, as cloud service providers offer flexible, scalable, and affordable computing resources on-demand, it is also becoming popular to deploy a big data processing cluster in the cloud. Although most of the deployments of a big data computing cluster are either local, or on the cloud, many organizations are also using a hybrid setup where both local and cloud resources are used together to form the cluster. However, it is challenging to schedule jobs in a cluster deployed on hybrid clouds while ensuring the SLA parameters such as monetary cost minimization, and deadline. In this chapter, we propose scheduling algorithms that can satisfy the SLA requirements of the jobs in a big data processing cluster deployed in a hybrid-cloud.

We have chosen Apache Spark as our target big data processing platform as it is vastly replacing traditional Hadoop-based platforms. Spark can utilize memory to store intermediate results to speed up the processing. Moreover, it is more scalable than other platforms and more suitable for running complex analytics jobs. Spark programs can be written in many high-level programming languages, and it also supports diverse data sources such as HDFS [11], Hbase[14], Cassandra[15] and Amazon S3. The data abstraction of Spark is called Resilient Distributed Dataset (RDD) [16], which is fault-tolerant. When a Spark job is launched, it creates one or more executors that use a fixed chunk of resources in any cluster nodes. These executors are used by a job to run multiple tasks in parallel at different stages of the data processing pipeline to work on various partitions of the dataset.

The default scheduler of Spark is FIFO, which schedules the jobs on a first-come-first-serve basis. The executors from a job are distributed in different nodes in a round-robin fashion for balancing the cluster load and improve performance. In addition, it can also consolidate the core usages and minimize the total number of nodes used in the cluster. However, if the nodes (VMs) are deployed in a public cloud, distributing the executors across different VMs can be costly as most of the VMs will be always turned on. In addition, there will be free resources in these VMs in an off-peak period when not many jobs are running in the cluster at the same time. Furthermore, if a hybrid cloud setup is considered, challenges within inter-cluster scheduling exist which include: design

https://www.marketsandmarkets.com/PressReleases/hybrid-cloud.asp
https://aws.amazon.com/s3/
https://spark.apache.org/docs/latest/job-scheduling.html

issues for federated multi-cluster, latency issues between different regional sub-clusters, and locality of the data. There are numerous works on inter-cluster schedulers [79, 80], which focus on addressing these challenges from a performance standpoint. However, these schedulers do not consider the VM usage cost of the Spark cluster deployed in a hybrid cloud setup. In this chapter, we complement these works and address two key objectives for hybrid cloud scheduling: cost-minimization and deadline violation reduction. We propose scheduling algorithms which work on the cluster-scheduling level, and utilize the pricing models of different VM instance types in a hybrid cloud to effectively handle the following challenges:

- Performing cluster-level scheduling to make fine-grained decisions for executor placements on a hybrid cloud environment.
- Minimizing the deadline violations for the jobs in the cluster.
- Minimizing the monetary cost of using the Virtual Machines (VMs) of the whole cluster.

In summary, our work makes the following key **contributions**:

- We formulate an optimization problem for SLA-based scheduling of Spark jobs in a hybrid cloud.
- We propose two job scheduling algorithms. The first algorithm is a modified version of the First-Fit (FF) heuristic for solving bin packing problems. The second algorithm uses a greedy approach to iteratively find the cost-optimal placement for each executor of a job. Both algorithms can improve the cost-efficiency of a hybrid Apache Spark cluster.
- We develop an event-based simulator in Java which can be used to simulate, test, and compare different job scheduling policies.
- We implement both of the proposed algorithms on top of Apache Mesos [10] cluster manager with separate extendable modules. Therefore, the implemented system is pluggable to Mesos and can be easily deployed in a hybrid cloud setup.

- We conduct extensive experiments in both simulated and real environments. Furthermore, we use real applications and workload traces under different scenarios to showcase the superiority of our proposed algorithms over the existing approaches.

The rest of the chapter is organized as follows. In section 5.2, we present the background to different frameworks and also the architectural considerations for a hybrid cloud deployment. In section 5.3, we discuss the existing works related to this chapter. In section 5.4, we show the system model and formulate the scheduling problem. In section 5.5, we present the proposed algorithms. In section 5.6, we show the simulation experiment setup, baseline algorithms and experimental results for simulation-based experiments. In section 5.7, we showcase the implemented prototype system in real platforms, discuss the benchmark applications and real experimental cluster setup, and demonstrate the feasibility of the proposed algorithms with performance evaluation from real experimental results. Section 5.8 concludes the chapter and highlights future work.

5.2 Background

5.2.1 Apache Spark

As compared to the disk-based MapReduce tasks of a typical Hadoop system, Apache Spark allows most of the computations to be performed in memory and provides better performance for some applications such as iterative algorithms. The intermediate results are written to the disk only when it cannot be fitted into the memory. Spark uses *Resilient Distributed Datasets (RDD)* to hold data in a fault-tolerant way. Each job/application is divided into multiple sets of tasks called stages which are inter-dependant. All these stages form a directed acyclic graph (DAG) and each stage is executed one after another. In a typical Apache Spark cluster, applications are submitted through a cluster manager to run in the cluster. Spark supports *Apache Mesos*, or *Hadoop Yarn*, or *Kubernetes* as cluster managers to allocate resources among applications. In addition, its own default *Standalone* cluster manager is also sufficient to handle a production cluster. All these

cluster managers support both static and dynamic allocation of resources.

Workers are the physical/compute nodes of an Apache Spark cluster where one or more application processes can be created depending on the resource capacity. In cloud deployments, one or more worker nodes can be created inside each Virtual Machines (VM). A Spark cluster can have one or more worker nodes but there is only a single *Master* node that is responsible for managing the worker nodes. Each application in Spark has a *SparkContext* object in its main program (also called the *Driver Program*) which creates and maintains *Executor* processes on worker nodes. An application uses its own set of executors to run tasks in parallel, in multiple threads and to keep data in memory and storage. In addition, these executors live for the whole duration of that application. All the executors of the same application must be identical in size. Hence, they will have the same amount of resources (CPU cores, memory, disk). There are two benefits of isolating applications from each other. First, a driver program can independently schedule its own tasks in the acquired executors. Second, each worker can have multiple executors from different applications running in their own JVM processes.

5.2.2 Apache Mesos

Apache Mesos is considered to be a data-center level cluster manager due to its capability of efficient resource isolation and sharing across distributed applications. In Mesos, jobs/applications are called frameworks and multiple applications from different data processing frameworks like Spark, Storm, and Hadoop can run in parallel in the cluster. Mesos introduces a novel two-level scheduling paradigm where it decides a possible resource provisioning scheme according to the weight, quota or role of a framework and offers resources to it. The framework's scheduler is responsible for either rejecting or accepting those resources offered by Mesos according to its scheduling policies. Mesos provides HTTP APIs to control the resource provisioning and scheduling of the whole cluster. Mesos supports dynamic resource reservations, thus resources can be dynamically reserved in a set of nodes by using the APIs and then a job/framework can be scheduled only on those resources. When a job is completed, resources can be taken

http://mesos.apache.org/documentation/latest/operator-http-api/

back and reserved for any future job. It is a significant feature of Mesos as any external scheduler implemented on top of Mesos can have robust control over the cluster resources. Furthermore, the external scheduler can perform fine-grained resource allocation for a job in any set of nodes with any resource requirement settings. Lastly, various policies can be incorporated into an external scheduler without modifying the targeted big data processing platform or Mesos itself; so the scheduler can be extended to work with other big data processing platforms. For the benefits mentioned above, we have built a prototype system on top of Mesos to implement our proposed scheduling algorithms. However, the proposed scheduling algorithms can be plugged to work with other modern cluster managers, such as Kubernetes, which also supports fine-grained resource allocation for containers (e.g, pods from Kubernetes terms).

5.2.3 Scheduling Levels

From the above discussion, we can observe that there are two levels of scheduling in the cluster. These are (1) Cluster Level: decision to select an appropriate VM to create an executor for a Spark job. From the cluster manager perspective, a container can be created and allocated with a fixed set of resources and then this container can be assigned to a job's executor. (2) Application Level: The Spark application driver process is responsible for scheduling tasks in the provisioned executors for a job. This scheduler should consider the locality of the data to improve the performance of a job. In this chapter, we work on the cluster level to decide in which VM each executor of a job should be created so that we can optimize the overall cluster usage cost. In addition, we also consider the deadline constraint to prioritize jobs with tight deadlines. As our proposed approaches work on a higher level, it can be applied to the Hadoop jobs as well. For example, a cluster manager such as Mesos supports jobs from different types of frameworks such as Hadoop and Spark. Thus, the proposed scheduling algorithms can be extended to support Hadoop jobs, where each Mesos container should be provisioned for a map or a reduced task.

5.2.4 Hybrid Cloud Deployment

There are different architectural considerations regarding the deployment of a hybrid cloud. For example, in a true multi-cluster setup, all the executors of a job should be placed in the same cluster. However, in a multi-cluster federation, there is a central point of control and the same job's executors can be distributed across multiple clusters. The latter approach may result in locality and latency issues, as the executors from the same job have to communicate over different regional boundaries. However, in a multi-cluster federation, there is only one cluster (although over multiple regions) from the job's perspective. In addition, there is more room for cost-efficiency as it is possible to squeeze out the free resources in the cheapest VMs across multiple clusters. Thus, in this chapter, we choose a federated multi-cluster setup, where a central Mesos cluster manager is responsible to manage all the VMs across two different regions. The Mesos cluster manager is deployed in the local region to work as the central point of control. In addition, the external scheduler and other resource reservation modules are also run locally for faster communication with the cluster manager. Although there can be latency and performance issues caused by this setting, we try to capture these issues in the system model by considering the increase in job completion times caused by these issues.

5.3 Related Work

The default framework scheduler for Spark is FIFO, which places the executors of a job in a round-robin fashion to balance the load in the cluster and improve performance. In addition, it can also consolidate the core usage to minimize the total nodes used in the cluster. However, it does not consider the pricing model of VM instances in either a single or a hybrid cluster setup. Fair and DRF [17] based schedulers can be used to improve the fairness among multiple jobs in a cluster, but they do not improve the cost-efficiency of the cluster.

There is some existing research for SLA-based job scheduling, which only focuses

https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/FairScheduler.html

on Hadoop MapReduce based jobs. Hwang et al. [49] proposed a resource provisioning model that can minimize the VM cost for deadline-constrained MapReduce applications in cloud. Mashayekhy et al. [26] proposed a greedy algorithm that finds the assignments of the map and the reduce tasks in machine slots to minimize the energy consumption of a Hadoop cluster. Nayak et al. [38] proposed a negotiation-based adaptive scheduler for scheduling Hadoop jobs in cloud. Cheng et al. [81] have considered future resource availability to improve job performance and reduce job deadline violations. Zeng et al. [82] proposed a greedy algorithm that reduces the monetary cost of using the public cloud while satisfying job deadlines. ChEsS [39] is a Pareto-based job-to-cluster assignment framework for cost-effective job scheduling across multiple MapReduce clusters. However, most of these works either consider a single cluster setup or tries to improve job performance. Moreover, these approaches are applicable to Hadoop jobs only as the architecture paradigm of Hadoop is different from Spark.

There are a few works that tried to improve different aspects of scheduling for Spark-based jobs. Sparrow [27] tried to improve the performance of the default Spark scheduling by using a decentralized, randomized sampling-based scheduler. Wu et al. [83] proposed a framework that provides the capability to perform large-scale data analytics across multiple-clusters. Maroulis et al. [34] provided an energy-efficient scheduler that uses the DVFS technique to tune the CPU frequencies for the workloads to reduce energy consumption. However, as our main target is cost-effectiveness, this approach can not be applied to our problem. Li et al. [84] also provided an energy-efficient scheduler. However, it does not consider cost as an objective. In addition, the algorithm assumes each job has the same executor size, which is equal to the total resource capacity of a VM. However, in reality, each job can have different resource requirements, and the VM instance size can also vary. Liu et al. [85] proposed a hierarchical multi-cluster big data framework for Apache Spark, which only focuses on improving job performance when the cluster is deployed in a hybrid-cloud. However, they do not consider any cost-efficiency in these clusters, and job deadlines. Sidhanta et al. [86] provided a mathematical model to estimate job completion times of a Spark job given its input size, iteration, and job type. In addition, they provide an optimal cluster composition technique which utilizes the default FIFO scheduler. However, this work does not consider different VM

pricing models in a hybrid-cloud setup. In addition, it is assumed that each job has the same executor size, which is the total resource capacity of a VM. However, we model the executor sizes at a more fine-grained level, so that multiple executors from one or more jobs can be co-located inside a single VM.

MCTE [87] is a cloud task scheduling strategy to minimize the task completion time and execution cost for the smart grid cloud. However, this work did not consider a hybrid cloud setup and cost minimization as an objective. AsQ [88] is also a task scheduling algorithm that places the task in either local or cloud VMs. Peláez et al. [89] introduced the problem of managing virtual machines and scheduling jobs in a cost-efficient way while meeting the deadlines. In the bag of tasks model, the tasks are independent of each other so the run-time of an individual task does not depend on whether another task from the same bag is placed in the cloud or local VM. However, in our work, we focus on the cluster-level scheduling where an executor runs one or more interdependent tasks that follows a DAG model.

If a cluster is deployed in a hybrid cloud, some of the VMs reside in the local premises and the rest of the VMs are hired from a cloud service provider. Thus, the cloud portion of the cluster can be considered to be in a different region. Therefore, challenges within inter-cluster scheduling exist which include: choosing a proper federated multi-cluster setup that determines how the clusters should be managed, increased latency between different executors deployed in different regions, and locality of the data required for a job. There are numerous works on inter-cluster schedulers, e.g., Yarn Federation, Kubernetes Federation, Medea [79] and Hyrda [80], which focused on addressing these challenges with objectives to improve the overall performance of the production cluster. Because for multiple regional clusters, it is more critical to focus on performance improvement and load-balancing. However, if a hybrid cloud setup is created with the use of public cloud VMs, minimizing cluster resource usage cost should be a key objective, along with maintaining an acceptable performance for the applications.

In summary, most of the existing approaches focus mainly on performance improvement. In addition, they do not consider a fine-grained level of executor placement while

https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/Federation.html

https://kubernetes.io/blog/2018/12/12/kubernetes-federation-evolution/

scheduling jobs. In contrast, our approach guarantees to launch a job on its required resources, tries to minimize deadline violations, can handle different sizes of executors of jobs and different VM instance sizes, and can reduce the overall cost of VM usage of a hybrid cloud deployed cluster by utilizing different pricing models.

5.4 SLA-based Job Scheduling

In this section, we describe the hybrid cloud model and formulate the problem of dynamic job scheduling between local VMs and cloud VMs. Major notations and descriptions presented in this chapter are listed in Table 5.1.

5.4.1 System Model

When a hybrid cloud setup is considered, both local and cloud VM instances can be chosen to be identical in resource capacity. However, when the target objective is to reduce cost, having a setup with different types of VM instances is more cost-effective, because jobs with fewer resource requirements can be fitted into small VMs to optimize cost. In addition, if the local part of the cluster is made with commodity resources, it is not possible to create similar VM instances with a set of heterogeneous physical hosts. Therefore, to tackle the scheduling problem more efficiently, the scheduler has to consider different VM instance sizes (as depicted in Fig. 5.1) to optimize cost. We consider a federated multi-cluster deployment where the cluster manager is the central point of control. The cluster manager controls both the local and the cloud VMs. The resource managers track the resource availability of the cluster and dynamically feed the updated status of the cluster to the scheduler. Thus, the scheduler has to match the resource requirement of the jobs with the resource availability in the cluster while trying to meet the target objectives. In our implemented prototype, we deploy the external scheduler, both of the resource managers, and the cluster manager in the local cloud.

In an Apache Spark cluster, each job consists of a set of executors with the same resource requirement. Furthermore, each VM/worker node has a set of available resources (e.g., CPU and memory) which can be used to place executors. However, executors from

Table 5.1: Definition of Symbols

Symbol	Definition
J	The current job to be scheduled
E	Total executors required for J
ξ	The index set of all the executors of J, $\xi = \{1, 2, 3, ..., E\}$
T_C^L	Profiled completion time for J for local-only placement of executors
T_C^H	Profiled completion time for J for hybrid placement of executors
T_C	Estimated completion time for J
T_D	Deadline for J
T_A	Arrival time for J
T_S	Start time for J
T_W	$T_S - T_A$, waiting time for J
M	The total number of local VMs
N	The total number of cloud VMs
δ^L	The index set for all the local VMs; $\delta^L = \{1, 2, ..., M\}$
δ^C	The index set for all the cloud VMs; $\delta^C = \{1, 2, ..., N\}$
P_j^L	The Price for a local VM; $j \in \delta^L$
P_j^C	The Price for a cloud VM; $j \in \delta^C$
C_j^L	Available CPU in a local VM, $j \in \delta^L$
C_j^C	Available CPU in a cloud VM, $j \in \delta^C$
M_j^L	Currently available Memory in a local VM, $j \in \delta^L$
M_j^C	Currently available Memory in a cloud VM, $j \in \delta^C$
C_i^τ	CPU demand of any executor of J, $i \in \xi$
M_i^τ	Memory demand of any executor of J, $i \in \xi$
t_j^L	Remaining active time for a VM before placing executor(s) of J, $j \in \delta^L$
t_j^C	Remaining active time for a VM before placing executor(s) of J, $j \in \delta^C$
Δt_j^L	Change in remaining active time after executor(s) of J is placed, $j \in \delta^L$
Δt_j^C	Change in remaining active time after executor(s) of J is placed, $j \in \delta^C$

different jobs can have different sizes. For example, suppose the CPU and memory requirements of an executor of job-1 are 2 cores and 4GB, respectively. Thus, if job-1 has 5 executors, all the executors must follow this resource requirement (e.g., 2 cores and 4GB memory). However, job-2 can have different resource requirements for its executors. For example, 4 cores, and 8GB of memory for each executor, which is different from the size of the executors from job-1.

For each submitted job in the cluster, the main problem is to find the mapping of all its executors to one or more available VMs. Besides, the combined resource requirements of all the placed executors in a VM is bound by its resource capacity. Therefore, resource constraints in each VM must be met while making any scheduling decisions. We consider a multi-tenant case where multiple jobs from different users can run on the cluster at the same time. Thus, if one or more executors from different jobs are placed in the same VM, then the resource capacity constraints of that VM must be satisfied by considering all the different sizes of executors from multiple jobs. This problem can be simplified by tracking the resource availability of VMs dynamically. Thus, the resource availability of the VMs can be presented to the scheduler, instead of the resource capacity. Initially, the resource capacity and resource availability of a VM will be the same. Although the resource capacity of a VM is always fixed, the resource availability of a VM will be reduced over time if one or more executors from one or more jobs are placed in it. In addition, if one or more jobs complete execution that had executor(s) in this VM, then the resource availability of the VM will be increased.

We consider the resource requirement of an executor in two dimensions – CPU cores and memory. Suppose, we are given a job with E executors where each executor has CPU and memory requirements of τ_i^{cpu} and τ_i^{mem}, respectively. Furthermore, each job has a deadline that needs to be met by the scheduler. After handling all the constraints, the scheduler should try to reduce the resource (or VM) usage cost of the cluster. In our case, we have a hybrid cloud setup where some VMs are located in local-premises and some VMs are hired from the cloud on a pay-per-use basis. We assume that if the resource requirements are met, the performance of all the executors from the same job is similar regardless of whether they are placed on local or cloud VMs.

Suppose J is the current job to be scheduled in the cluster. If one or more previous

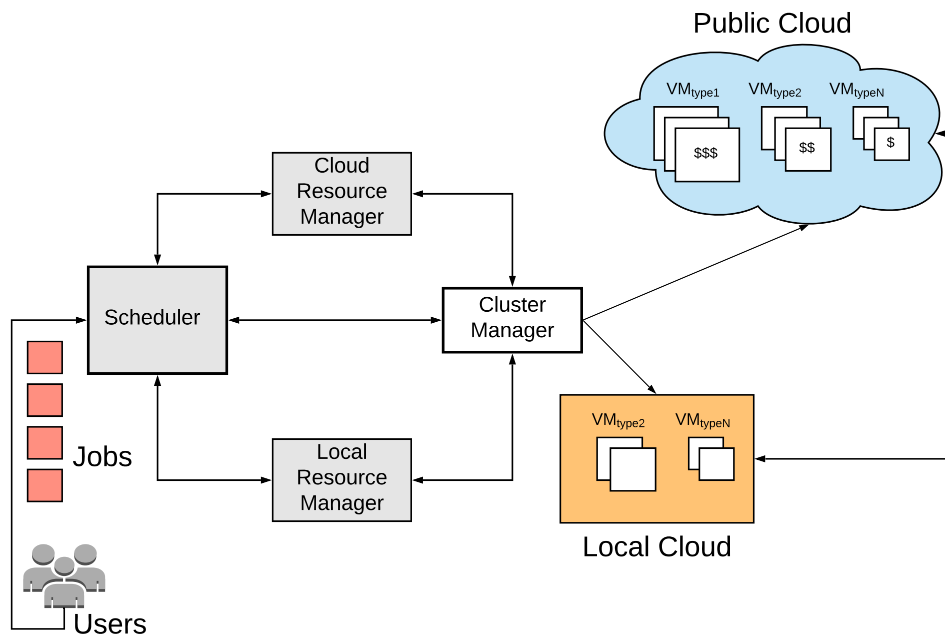


Figure 5.1: Proposed Hybrid Cloud Model. The resource managers (cloud and local) are controlled by the scheduler to create executors of job in VMs, turn on/off VMs, and to monitor the cluster states.

jobs are still running in the cluster, the scheduler has to make a decision on whether to utilize the spare resources on the already active VMs to place one or more executors of J, or turn on new local/cloud VMs. Therefore, to make a cost-optimal scheduling decision for each job, the scheduler should use a combination of both local/cloud VMs.

In our proposed model, the scheduler uses a queue which follows the EDF (earliest deadline first) order of jobs, to reduce deadline violations. The scheduler iterates over each job, dynamically observes the latest cluster resource availability, and makes scheduling decisions to place the executors for that job. For simplicity, we present the model on a per-job basis, which means the model represents what the scheduler observes for making decisions for the next job in the scheduling queue. In the following subsections (5.4.2, 5.4.3), we model the cost and resource constraints for both local and cloud VMs. Then in subsection 5.4.4, we combine the resource models and constraints to formulate the scheduling problem.

5.4.2 Local Resource Model

Definition-1 (Local VM Set): Consider a set $\delta^L = \{1, 2, \cdots, M\}$, where M is the total number of local VMs, $1 \leq j \leq M$ is the jth VM deployed locally.

The expression of local cost for the current job, J is derived as follows:

$$Cost^L = \sum_{j \in \delta^L} x_j \times P_j^L \times \Delta t_j^L, \tag{5.1}$$

where P_j^L is the unit price for a local VM; we define a binary decision variable x_j to indicate whether a local VM is active or not, i.e.,

$$x_j = \begin{cases} 1 & \text{if } \sum_{i \in \xi} u_{ij} > 0; \\ 0 & \text{otherwise.} \end{cases} \tag{5.2}$$

where we define a binary decision variable u_{ij} to indicate whether executor i is placed in a local VM j or not, i.e., $\forall j \in \delta^L$, we have

$$u_{ij} = \begin{cases} 1 & \text{if executor } i \text{ is placed in the local VM } j; \\ 0 & \text{otherwise.} \end{cases} \tag{5.3}$$

Δt_j^L is the change in the remaining active time for a local VM j if any executor of J is placed in it, which is calculated by:

$$\Delta t_j^L = \begin{cases} (T_C - t_j^L) & \text{if } T_C > t_j^L; \\ 0 & \text{otherwise.} \end{cases} \tag{5.4}$$

Here, t_j^L is the remaining active time for a local VM before placing any executor of the current job. T_C is the estimated completion time of the current job, J. We assume that the T_C can be provided for each job, which is generally measured from the job profile information. Now, the executors of the job can be placed only on the local VMs, or in a hybrid manner where both local and cloud VMs can be used. However, if any cloud VMs are used for executor placements, T_C will be higher than local-only placements, due to local

to Cloud data transmissions and network latency. Suppose, the job is profiled in both settings, and T_C^L indicates the profiled completion time for the job for local-only placement. In addition, T_C^H indicates the profiled job completion time for a hybrid setting. Thus, for the local model, T_C can be defined as:

$$T_C = \begin{cases} T_C^L & \text{if } \sum u_{i,j} == E \;\; \forall i \in \xi, \forall j \in \delta^L; \\ T_C^H & \text{otherwise.} \end{cases} \tag{5.5}$$

Here, E is the total number of executors required by the job, so if the summation of all the local placements equals E, it indicates that the job will be running entirely in the local VMs.

Furthermore, the total resource demands of all the executors placed in a VM should not exceed the total resource capacity of that VM. Note that, this can be done simply if the current resource availability of the VM is checked against the resource demands of executor(s) of the current job. Suppose, C_i^τ and M_i^τ are the CPU and memory resource demands for each executor of the current job, respectively. Thus, the resource constraints for local VMs must be satisfied as follows:

$$\sum_{i \in \xi} (u_{ij} \times C_i^\tau) \leq x_j \times C_j^L, \qquad \forall j \in \delta^L, \tag{5.6}$$

$$\sum_{i \in \xi} (u_{ij} \times M_i^\tau) \leq x_j \times M_j^L, \qquad \forall j \in \delta^L. \tag{5.7}$$

where C_j^L and M_j^L are the currently available CPU and memory resources in the local VM j, respectively. Therefore, the scheduler can choose to place one or more executors from the current job in the same VM if the current resource availability permits.

5.4.3 Cloud Resource Model

Definition-2 (Cloud VM Set): Consider a set $\delta^C = \{1, 2, \cdots, N\}$, where N is the total number of cloud VMs, $1 \leq j \leq N$ is the jth VM deployed on the cloud.

Similarly, the expression of cloud cost for the current job, J is derived as follows:

$$Cost^C = \sum_{j \in \delta^C} y_j \times P_j^C \times \Delta t_j^C, \tag{5.8}$$

where P_j^C is the unit price for a cloud VM; we define a binary decision variable y_j to indicate whether a cloud VM is active or not, i.e.,

$$y_j = \begin{cases} 1 & \text{if } \sum_{i \in \xi} v_{ij} > 0; \\ 0 & \text{otherwise.} \end{cases} \tag{5.9}$$

where we define a binary decision variable v_{ij} to indicate whether executor i is placed in a cloud VM j or not, i.e., $\forall j \in \delta^C$, we have

$$v_{ij} = \begin{cases} 1 & \text{if executor } i \text{ is placed in the cloud VM } j; \\ 0 & \text{otherwise.} \end{cases} \tag{5.10}$$

Δt_j^C is the change in the remaining active time for a cloud VM if any executor of the current job is placed in it, which is calculated by

$$\Delta t_j^C = \begin{cases} (T_C^H - t_j^C) & \text{if } T_C^H > t_j^C; \\ 0 & \text{otherwise.} \end{cases} \tag{5.11}$$

where T_C^H is the estimated completion time of the current job, when one or more cloud VMs are used; and t_j^C is the remaining active time for a cloud VM before placing any executor of the current job. Further, the resource constraints for cloud VMs must be satisfied as follows:

$$\sum_{i \in \xi} (v_{ij} \times C_i^\tau) \leq y_j \times C_j^C, \qquad \forall j \in \delta^C \tag{5.12}$$

$$\sum_{i \in \xi} (v_{ij} \times M_i^\tau) \leq y_j \times M_j^C, \qquad \forall j \in \delta^C \tag{5.13}$$

On the one hand, because the total number of the local VMs might be limited, we can

use the cloud VMs for computing. Therefore, we can assume that $M < N$. On the other hand, however, the usage cost of the VMs in local VMs is usually lower than that on the cloud; hence we can assume that $P_j^L < P_j^C$. Therefore, similar VM instances deployed in local-premises cost lower than cloud VM instances.

5.4.4 Problem Formulation

Based on the system model, we now formulate the job scheduling problem to minimize the cost of using the whole cluster while scheduling the current job. The total cost is modeled as the aggregated cost of using all the VMs from both local and cloud.

Executor Placement Constraint: An executor can be placed only in one of the VMs and this placement constraint is denoted as:

$$\sum_{j \in \delta^L} u_{ij} + \sum_{j \in \delta^C} v_{ij} = 1, \qquad \forall i \in \xi. \tag{5.14}$$

Resource Capacity Constraints: The total resource demands of all the executors placed in a VM should not exceed the total resource capacity of that VM. These constraints are described in (5.6), (5.7), (5.12) and (5.13).

Job Deadline Constraint: If the job deadline is considered, whether a job fails to complete before the given deadline can be predicted by using Eq. 5.15.

$$T_C < T_D - T_W, \tag{5.15}$$

where $T_W = T_S - T_A$ is the waiting time for the current job to be scheduled. Note that, if the executors are not placed entirely in the local VMs, then T_C will be set to T_C^H in the local resource model.

On the one hand, if the job deadlines are not considered in the scheduling algorithm, a job which is predicted to fail will be scheduled, only to waste resources which could be used by any future job to successfully complete before their deadlines. On the other hand, if a job is predicted to violate its deadline, it can be discarded without passing to the scheduling algorithms. Thus, more resources will be freed to ensure that more jobs can be successfully finished before the deadline. In the experiment section, we show the

impact on deadline violations by the scheduling algorithms for both cases.

Therefore, the job scheduling problem can be formulated as Cost-Min:

$$\begin{aligned} \min : Cost^{total} &= Cost^{L} + Cost^{C}, \qquad (5.16) \\ s.t. \ &: (5.6), (5.7), (5.12), (5.13), (5.14), (5.15). \end{aligned}$$

The above problem is mixed-integer linear programming (MILP) [90] and non-convex [91], generally known as NP-hard problem [92]. The computational complexity will significantly increase due to the binary variables.

5.5 Proposed Job Scheduling Algorithms

We try to maximize the deadline met percentage by two ways: (1) by following an Earliest Deadline First (EDF) order to schedule jobs, so that if multiple jobs are waiting to be scheduled at the same time, jobs with tighter deadlines will have higher priority, and (2) before passing the job specifications to the scheduler, we utilize a job's completion time estimate (T_C) to check whether the job has a chance of violating the deadline. If so, we remove this job from the queue and do not schedule it. In this way, we keep some resources free in the cluster for future jobs to increase the overall deadline met numbers. The job queue is maintained externally from the scheduling algorithm, along with the cluster resource availability. Both the job queue and the cluster states are updated dynamically. Only the current job's specification and the cluster states are passed to a scheduling algorithm to make placement decisions. In this way, we reduce the overhead on the scheduling algorithm. If it is estimated that the job will be completed before the deadline, it is passed to the scheduler to make cost-effective executor placement decisions. We propose two algorithms to solve the scheduling problem. The first algorithm is a modified version of the First Fit (FF) heuristic algorithm for bin packing optimization problem. The second algorithm has a greedy approach and iteratively places all the executors of a job in the most cost-optimal position.

5.5.1 First Fit (FF) Heuristic-based Algorithm

In the bin packing problem, items of different volumes must be packed into a finite number of bins or containers each of a fixed given volume in a way that minimizes the number of bins used. In our case, we have a similar problem where the executors can be considered as the items which need to be packed into a finite number of VMs (bins). Thus, the scheduling problem formulated in Section 5.4.4 can be thought of as a two-dimensional (2D) vector bin packing problem, where each of the VM is a bin having two dimensions, i.e., CPU cores and memory. Each executor from a job has a fixed resource requirement in these two dimensions; thus, an executor can be thought of as an item. Therefore, the objective is to minimize the total number of bins (VMs) used to pack (place) a given set of items (executors) for each job. Algorithm 6 shows the modified version of the First Fit (FF) heuristic [75] algorithm, which can be used for executor placement in the scheduling process.

Algorithm 6: First Fit (FF) Heuristic Algorithm

Input: $Job\ \{E, \xi, C_i^\tau, M_i^\tau, T_C\}$: The current job to be scheduled, $ActiveVMList$: The list of all the active VMs (includes both cloud and local VMs)
Output: $PlacementList$, a list of VMs where the executors of Job will be placed

```
1  Procedure FF(Job, ActiveVMList)
2      PlacementList ← ϕ
3      forall vm ∈ ActiveVMList do
4          while Placement of an executor in vm satisfies all the resource constraints do
5              Update(vm)
6              PlacementList.add(vm)
7              if PlacementList.size = E then
8                  return PlacementList
9      if Cluster has unused VM(s) then
10         Turn on the cheapest vm_new that satisfies all the resource constraints of an
           executor
11         ActiveVMList ← ActiveVMList ∪ vm_new
12         goto step 3
13     return Failure
```

The input to this algorithm is the specification of the current job $(E, \xi, C_i^\tau, M_i^\tau, T_C)$ to be scheduled, and a list of currently active VMs (either in local or cloud) in the cluster.

The output is the *PlacementList*, which is a list of VMs where the executors of the current job should be placed. For each active VM, the algorithm first checks whether the placement of an executor of the current job will satisfy the resource constraints (lines 3-4). If so, the resource capacity of the current VM is updated (line 5), and the current VM is added to the *PlacementList*. The algorithm tries to place as many executors as possible in the same VM if the resource requirements are met. Otherwise, it tries the next active VM. If the total number of added VMs to the *PlacementList* reaches the total required number of executors for the current job, the algorithm returns with the placement list. If the currently active VMs are not sufficient to place any executor, then the cheapest VM is turned on (if available) and is added to the active VM list (lines 12-14). Then, steps 3-10 are repeated again. If the cluster does not have sufficient resources to place all the executors of the current job, the algorithm returns failure (line 17).

5.5.2 Greedy Iterative Optimization (GIO) Algorithm

The aforementioned MILP problem can be solved in polynomial time if the problem is relaxed from a per-job basis (finding the most cost-optimal placements of all the executors of the current job) to a per-executor basis (only find the most cost-effective placement of one executor from the current job). Although solving the relaxed problem will provide near-optimal results as compared to the original problem, it can be solved in polynomial time. We propose a greedy iterative optimization (GIO) algorithm, which utilizes the pricing model of different VM instances and the estimated completion time of each job to find cost-efficient executor placement (on a per-executor basis).

Suppose, the executor(s) from one or more jobs are running in a vm (deployed either in the cloud or in the local part of the cluster). Let T_{vm} be the active remaining time of the vm. If any executor of the current job J is placed in vm, the additional active remaining time of vm due to this placement is ΔT_{vm}, which can be found in Eq. 5.17.

$$\Delta T_{vm} = \max(0, T_C - T_{vm}). \tag{5.17}$$

Now, if the cluster has sufficient local resources to place all the executors from the current job, then T_C can be set to T_C^L, otherwise it can be set to T_C^H (Eq. 5.5). Hence, we

can calculate the cost incurred by placing an executor of J in vm by using Eq. 5.18.

$$Cost_e^J = \Delta T_{vm} \times P_{vm}. \tag{5.18}$$

Here, if vm is deployed locally, then $P_{vm} = P_j^L$ ($j \in \delta^L$). Otherwise, if vm is deployed on cloud, then $P_{vm} = P_j^C$ ($j \in \delta^C$). Suppose, vm is already in use and has some free resources to place one or more executors for the current job. If placing the new job's executor(s) in it does not make it run longer than before (if $T_C \leq T_{vm}$), or only makes it run further for a short period of time ($T_C - T_{vm}$ approaches 0), we can save cost by placing the current job's executor(s) in it.

Algorithm 7: Greedy Iterative Optimization (GIO) Algorithm

Input: Job $\{E, \xi, C_i^\tau, M_i^\tau, T_C\}$: The current job to be scheduled, $LocalVMList$: The list of all the local VMs, $CloudVMList$: The list of all the Cloud VMs

Output: $PlacementList$, a list of VMs where the executors of Job will be placed

```
1  Procedure GIO(Job, LocalVMList, CloudVMList)
2      VMList ← φ
3      if LocalAvailability(Job, LocalVMList) == true then
4          VMList ← LocalVMList
5      else
6          VMList ← LocalVMList ∪ CloudVMList
7      PlacementList ← φ
8      Sort(VMList) // Sort the VMs in an increasing order of Cost_e^J
           (Eq. 5.18)
9      forall vm ∈ VMList do
10         while Placement of an executor in vm satisfies all the resource constraints do
11             Update(vm)
12             PlacementList.add(vm)
13             if vm was unused then
14                 Turn on vm
15             if PlacementList.size == E then
16                 return PlacementList
17     return Failure
```

Algorithm 7 shows the proposed GIO algorithm. The input to this algorithm is the current job to be scheduled, and a list of all the local VMs, and the list of all the cloud

VMs. The output is the *PlacementList*, which is a list of VMs where the executors of the current job should be placed. At first, we check whether the current local resource availability is sufficient to place all the executors of the current job (line 3). If yes, we only utilize the local VMs (line 4), otherwise all the VMs (line 5). Then, the *VMList* is sorted in an increasing order of $Cost_e^J$ values (line 10). If the resource constraints are met, then the current *vm* is greedily used to place as many executors as possible (lines 12-14). If the currently chosen *vm* was inactive, it is turned on (lines 15-16). The steps for executor placement are repeated until all the executors of the current job are placed (lines 18-19). If the cluster does not have sufficient resources to place all the executors for the current job, a failure is returned (line 23).

Note that, for both FF and GIO algorithms, if there are not enough resources for the current job (a failure is returned by the algorithms), the scheduler will wait until more resources are freed so that it can schedule the current job.

5.5.3 Complexity Analysis

To calculate the worst-case time complexity of the proposed algorithms, we first assume that the total number of VMs in the cluster is m, which includes both cloud and local VMs. In the worst-case scenario, for every executor, the scheduler has to iterate through each and every VM to find its placement. Hence, if the current job's total number of executor requirement is e, the worst-case time complexity of Algorithm 6 is $\mathrm{O(me)}$. For Algorithm 7, the time required to check the local resource availability is m. In addition, the time required to sort the *VMList* (which may contain all the m VMs in worst-case) is $\mathrm{mlog(m)}$. Therefore, the worst-case time complexity of Algorithm 7 is $\mathrm{O(m + mlog(m) + me)}$.

5.6 Performance Evaluation - Simulation

We have used both simulation and real experiments to compare our proposed scheduling algorithms with the baseline algorithms. In this section, we discuss the experimental setup for simulation experiments, baseline scheduling algorithms used to compare our

Table 5.2: Simulation Cluster Details

Instance Type	CPU Cores	Memory (GB)	Quantity (small-scale)	Quantity (large-scale)
m1.large	4	16	Local=1; Cloud=2	Local=10; Cloud=50
m1.xlarge	8	32	Local=1; Cloud=2	Local=10; Cloud=50
m2.xlarge	12	48	Local=1; Cloud=2	Local=10; Cloud=50

Table 5.3: VM Instance Pricing Models

	Pricing Model 1		Pricing Model 2		Pricing Model 3		Pricing Model 4		Pricing Model (Real)	
Instance Type	Price (Cloud)	Price (Local)	Price (Cloud)	Price (Local)	Price (Cloud)	Price (Local)	Price (Cloud)	Price (Local)	Price (Cloud)	Price (Local)
m1.large	\$0.004/s	\$0.001/s	\$0.002/s	\$0.001/s	\$0.002/s	\$0/s	\$0.002/s	\$0.002/s	\$0.24/h	\$0.12/h
m1.xlarge	\$0.008/s	\$0.002/s	\$0.004/s	\$0.002/s	\$0.004/s	\$0/s	\$0.004/s	\$0.004/s	\$0.48/h	\$0.24/h
m2.xlarge	\$0.012/s	\$0.003/s	\$0.006/s	\$0.003/s	\$0.006/s	\$0/s	\$0.006/s	\$0.006/s	\$0.72/h	\$0.36/h

proposed algorithms, and the results from the simulation experiments.

5.6.1 Simulation Setup

Table 5.2 shows the simulation cluster details. We have used three types of VMs, each having different resource capacities. We have designed the clusters for both small-scale and large-scale experiments. Generally, we have more resources on the cloud than the local part of the cluster. Therefore, the small-scale cluster contains 3 VMs from each type of VM instance, where 1 VM is considered to be deployed locally, and 2 VMs are considered to be deployed on cloud. For the large-scale experiment, 60 VMs from each type of VM instance are used, where 10 VMs are considered to be deployed locally, and 50 VMs are considered to be deployed on cloud. We have designed different pricing models so that the effects of a pricing model in any scheduling algorithm can be evaluated. As shown in Table 5.3, the price of the same instance type in cloud is four times higher than local in pricing model 1, but only two times higher in pricing model 2. In pricing model 3, the price of using any local instance is 0. Lastly, in pricing model 4, the price of using the same type of instance is equal regardless of whether the instance is in cloud or locally deployed.

The job arrival times are generated from a Poisson distribution. We have designed our experiment to simulate both a high-load and a light-load period of the cluster. A Poisson mean of 5 and 100 is used to generate the job arrival rates for the high-load and light-load period, respectively. These mean values for Poisson distribution are chosen to reflect job arrival rates in real clusters in both low load and high load period, which is observed in Facebook Hadoop workload trace. The estimated job completion time for each job is generated using an exponential distribution with lambda ($\lambda = 0.01$). In addition, if a scheduler places the executors in a hybrid setting, where one or more executors are placed in the cloud VMs, then the simulation environment dynamically increases the job completion times by 30%. This is due to the fact that inter-cluster latency between executors and data locality issues will cause performance degradation for the jobs. A relaxed deadline for each job is generated by adding the job's estimated completion time with a threshold value (1000 seconds for the light-load period, 5000 seconds for the high-load period). All the resource requirements for each job are generated randomly within a range of 1-6 (for CPU cores), 1-10 (for memory in GB), and 1-8 (for total executors). All the simulation experiments are repeated 5 times to accommodate the randomness while calculating the statistics.

We have implemented an event-based simulator in Java to simulate the job scheduling in a hybrid cloud setup. We have implemented the proposed and baseline algorithms in this simulator to evaluate and compare them regarding different aspects. The simulator is open-source, and can be used to simulate new scheduling policies.

5.6.2 Baseline Schedulers

- **First in First out (FIFO):** It is used as a default scheduler in many big data processing frameworks including Apache Spark. Here, the executors of a job are placed in a round-robin fashion. However, as this default FIFO scheduler does not consider pricing models or different instance types in the hybrid cloud, resources are wasted if the cluster is not fully loaded with jobs.

https://github.com/SWIMProjectUCB/SWIM/wiki/Workloads-repository
https://github.com/tawfiqul-islam/RM-Simulator

- **First in First out Consolidate (FIFO-C):** Another round-robin approach used by the Spark scheduler to minimize the total number of VMs used. Note that, it works by packing executors on the already running VMs to avoid choosing the unused VMs.

- **AsQ [88]:** This scheduler addresses the task scheduling problem in hybrid cloud and has similar objectives as our work. AsQ considers the deadline constraint and tries to minimize the cost of the public cloud by maximizing the utilization of the private cloud. In addition, to avoid network latency issues between the public and private cloud, AsQ places the tasks for a job either in a local-only or in a cloud-only manner.

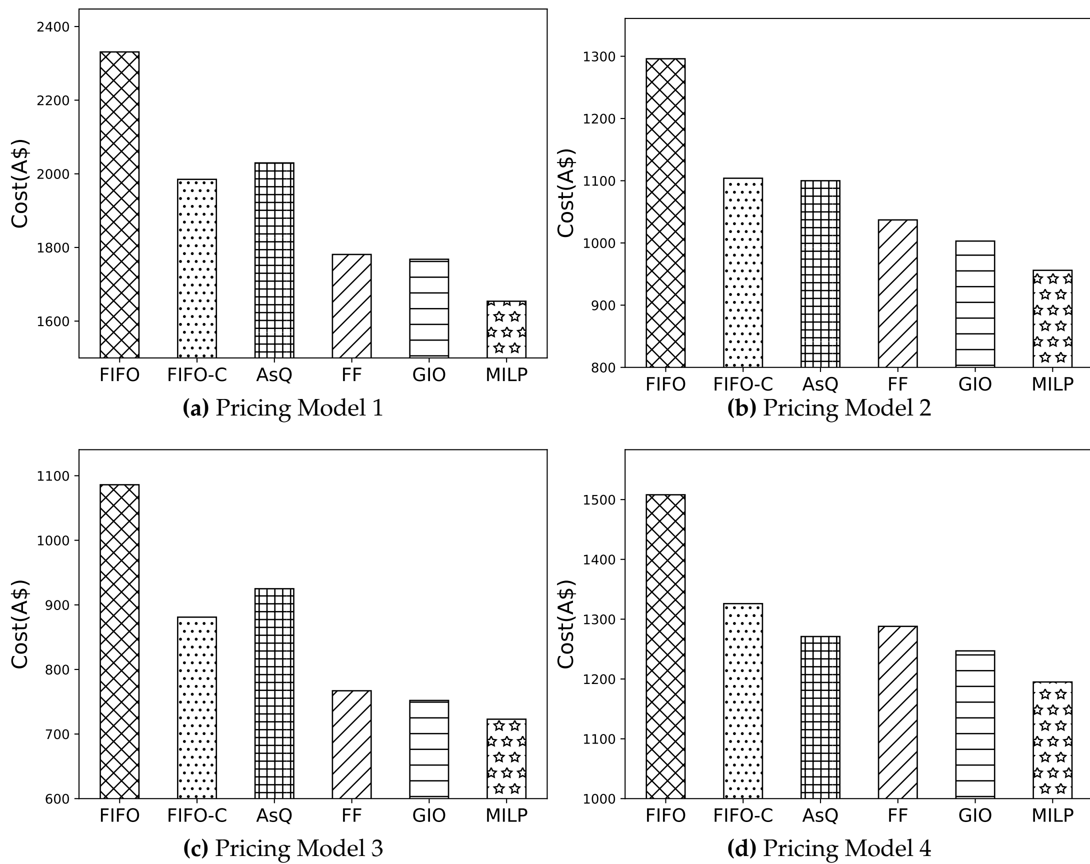
- **Mixed-Integer Linear Programming (MILP):** We have designed a MILP-based scheduler that generates the optimal cost-efficient placements for all the executors of each job. We have used SCP Solver API to solve the MILP problem in this scheduler. SCP solver uses a revised branch-and-cut [93] based approach for solving the MILP problem. However, the solver can take a significantly long time to solve the scheduling problem if the problem size is big (large cluster with many VMs, or jobs with many executors).

5.6.3 Simulation Results

In this subsection, we demonstrate the results from the simulation experiments with both small-scale and large-scale setups. However, as the MILP-based algorithm is not scalable and becomes infeasible when the problem size goes bigger, it is excluded from the large-scale simulation experiments. The small-scale experiment is used to compare the proposed algorithms with the baseline algorithms regarding cost-efficiency, scheduling overhead, and deadline violation. Furthermore, the large-scale setup is used to show the scalability of the proposed algorithms.

---

http://scpsolver.org/



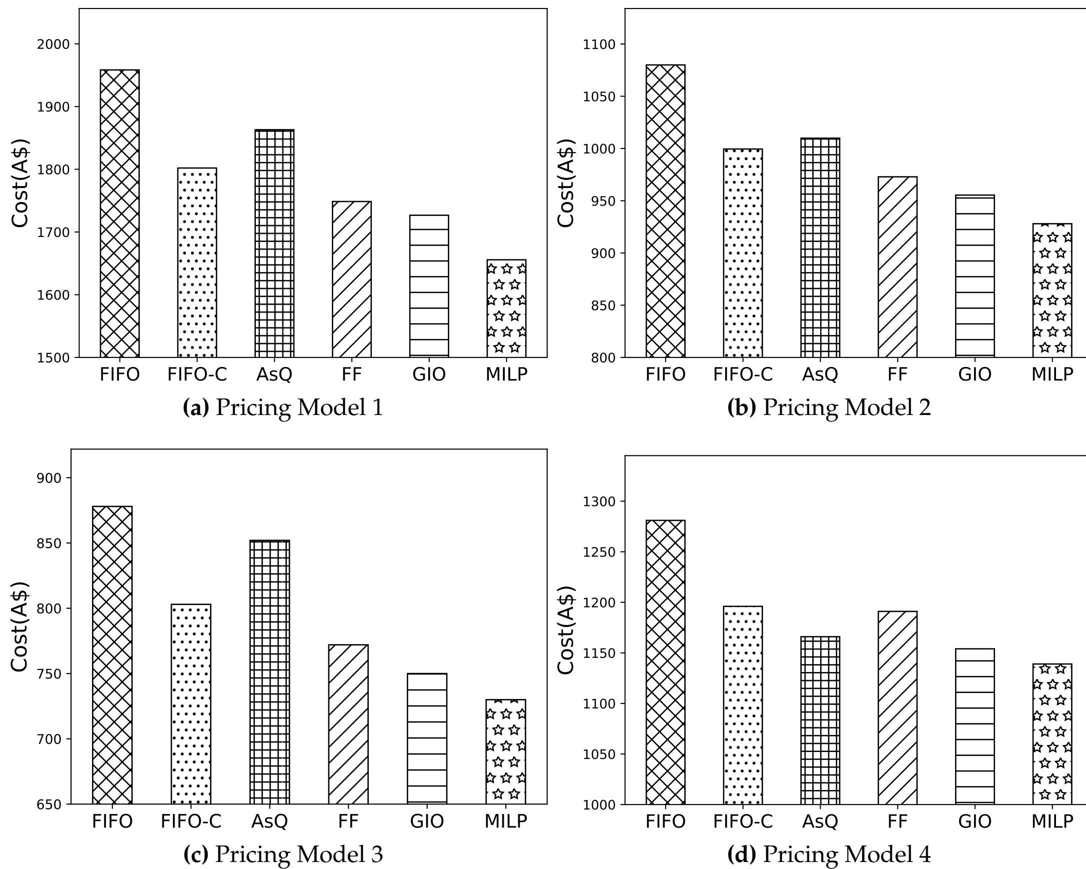
(a) Pricing Model 1

(b) Pricing Model 2

(c) Pricing Model 3

(d) Pricing Model 4

Figure 5.2: Cost comparison between the scheduling algorithms under different VM instance pricing models in a lightly loaded cluster (the lower the better). The scheduling delay is omitted to show how close the schedulers perform to the MILP solution regarding true cost calculation.



(a) Pricing Model 1

(b) Pricing Model 2

(c) Pricing Model 3

(d) Pricing Model 4

Figure 5.3: Cost comparison between the scheduling algorithms under different VM instance pricing models in a highly loaded cluster (the lower the better). The scheduling delay is omitted to show how close the schedulers perform to the MILP solution regarding true cost calculation.

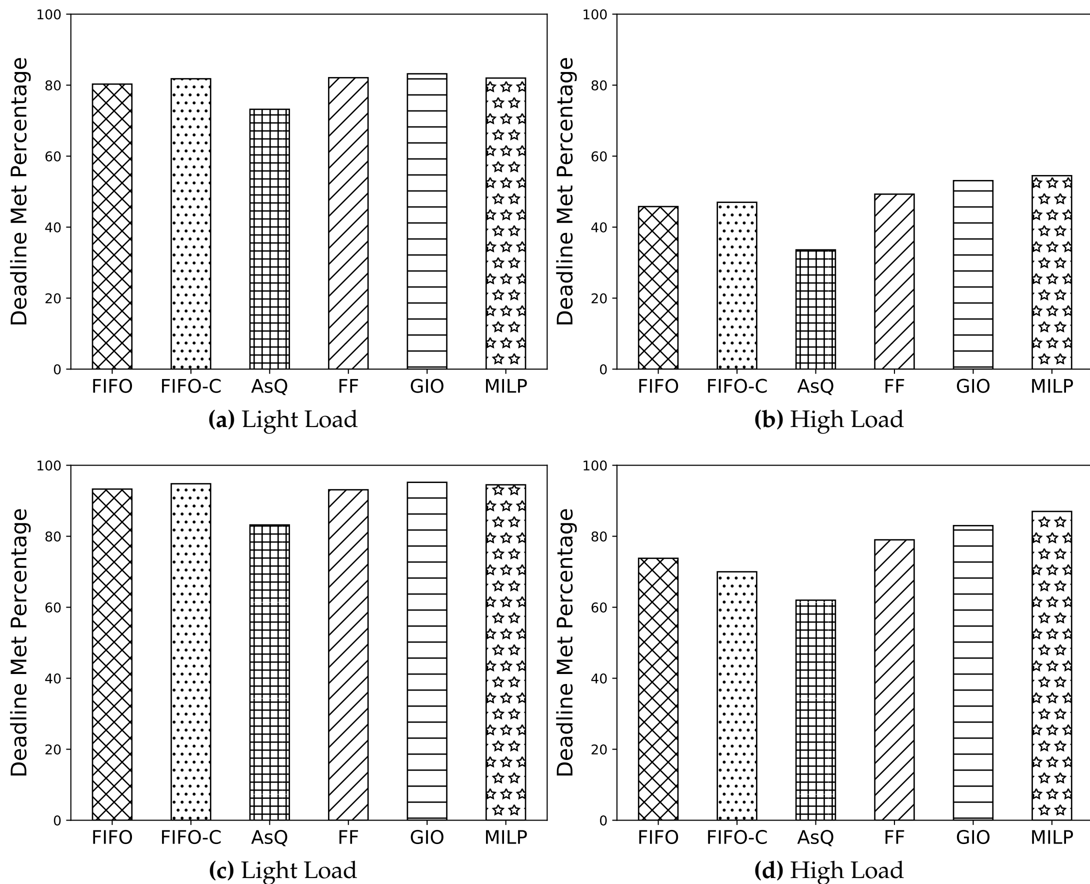


Figure 5.4: Comparison of deadline met percentage between the scheduling algorithms in both high load and light load period of the cluster (the higher the better). (a) and (b) shows the result when the deadline constraint is not considered. (c) and (d) shows the result when the deadline constraint is considered, and the jobs which are predicted to fail are removed from the job queue.

Evaluation of Cost Efficiency

In this evaluation, we have measured the cost of using the whole cluster to calculate the cost incurred by a specific scheduling algorithm. We save the turn-on or turn-off status of every VM in each second. Then we use one of the pricing models to calculate the cost incurred by using each VM during the whole scheduling process. Lastly, the total cost is calculated by summing up the cost of all the VMs. Note that, the MILP-based algorithm sometimes take exponential time to complete. Therefore, for fair cost comparison and to show how close the proposed schedulers performed to the MILP-based algorithm, the increased amount of VM usage cost due to the scheduling overhead is excluded. Figs. 5.2 and 5.3 depict the comparison of cost between different scheduling algorithms under different pricing models in both lightly loaded and highly loaded clusters, respectively. It can be observed that, under any pricing models, the proposed FF and GIO scheduling algorithms significantly reduce the cost usage of the cluster than the default FIFO and FIFO-C scheduling algorithms. The GIO scheduling algorithm can reduce the cost up to 25%, whereas the FF scheduling algorithm can reduce the cost up to 15% than the FIFO and FIFO-C algorithms. Although FIFO-C utilizes a round-robin approach, it tries to do so in the active VMs only. Thus, this approach reduces the cost as compared to the naive FIFO. The AsQ algorithm only places the executors from the same job either in a local-only or cloud-only fashion. However, the proposed FF and GIO algorithms utilize both cloud and local VMs, thus, can reduce the cost further. The FF algorithm starts the cheapest VM when the current set of VMs do not have sufficient resource capacity to schedule a new job. When placing executors, it does not consider VM prices and job runtimes in VMs, but selects the first available VM which satisfies the resource constraints. However, as the GIO algorithm takes the job duration and pricing models into consideration, it always performs slightly better than the FF. In addition, it considers network latency and data transmission issues into consideration, and only goes for a hybrid placement if there are not sufficient local resources available. However, even for hybrid placement, it uses the spare resources from both local and cloud VMs to reduce cost significantly. As the MILP algorithm solves the scheduling problem optimally before placing the executors of each job, it provides the most cost-efficient solution. However, both FF and GIO algorithm reduces the cost significantly and operates

very close to the ILP solution. Both algorithms only incur 8%-10% more cost than the ILP algorithm under different pricing models in both lightly loaded and highly loaded clusters.

Evaluation of Job Deadline

This evaluation is done by taking the percentage of jobs that finish before the given deadline. We have done experimentation in two cases. In the first case, we have recorded the deadline met percentage when all the algorithms do not use deadline as a constraint. In the second case, all the algorithms consider the deadline as a constraint, and if it can be predicted from the job estimation time that a job is going to fail to meet its deadline, that job is not scheduled. The reason to conduct experiments in both cases is to observe the effects of freeing up resources from the failed jobs (estimated), which creates more room for future jobs so that they can meet the deadline.

Figs. 5.4a and 5.4b depict the deadline met percentage by all the scheduling algorithms in light load and high load clusters, respectively, when the deadline is not used as a constraint. The deadline met percentage is lower in case of high load scenarios as the cluster is over-utilized, and there is a shortage of resources which causes many jobs to violate the deadline. For the light load case, the deadline met percentage is higher as there are more resources to accommodate the jobs whenever they arrive. In both cases, the MILP algorithm performs the best as it creates the least amount of resource fragments by tightly packing the executors. However, the FIFO algorithm distributively places executors that create many resource fragments in the cluster, which causes resource scarcity and more deadline violations. The FIFO-C algorithm performs slightly better than FIFO due to the consolidated approach. However, the AsQ algorithm chooses either local-only or cloud-only mode for placement. Thus, when the cluster is overloaded with many jobs at the same time, there is an increase in deadline violations due to resource scarcity. Both the proposed algorithms perform closely to the MILP-based algorithm where the GIO and FF algorithms are behind in the deadline met percentage by 5% and 8%, respectively. Figs. 5.4c and 5.4d exhibit the deadline met percentage by all the scheduling algorithms in both light load and high load clusters when the deadline

constraint is used. It can be observed that, as many predicted to be failed jobs are not scheduled in the cluster, the overall deadline met percentage improved significantly for all the scheduling algorithms. The MILP-based algorithm performs the best in this case as well, followed by the GIO and FF algorithm, while the AsQ performs the worst.

Evaluation of Scheduling Delay

The scheduling delay is the time an algorithm takes to make scheduling decisions for all the executors of a job. We have measured it by measuring the time it takes from calling a particular scheduling algorithm up to the return from the scheduling algorithm with all the executor placement decisions for a job. The average scheduling delay for an algorithm is calculated by taking the average of the scheduling delays for all the jobs scheduled by that algorithm.

Table 5.4: Average Scheduling Delay (small-scale)

Algorithm	**Average Scheduling Delay**
FIFO	0.18 μs
FIFO-C	0.20 μs
AsQ	0.31 μs
FirstFit	0.28 μs
GIO	0.40 μs
ILP	1.85 s

Table 5.4 shows the average scheduling delay by each algorithm in the small-scale setup. As the FIFO and FIFO-C algorithms follow a round-robin approach while placing the executors, the decision time is the shortest. Thus these algorithms have the lowest scheduling overheads. AsQ, FF and GIO are heuristic-based approaches, so these algorithms also showcase low scheduling delays which are closed to the native schedulers (FIFO and FIFO-C). However, the MILP-based solution takes as long as 10-minutes in the worst-case even in the small-scale cluster setup and has an average scheduling de-

lay of 1.85 s. Therefore, even though this algorithm can find the optimal cost-efficient executor placements, it is not scalable. Thus, it is only applicable to small-scale clusters.

Evaluation of Scalability

We have performed simulation on a large-scale setup where the cluster has 60 VMs (10 local VMs and 50 cloud VMs). We simulated the scheduling of 10,000 jobs in one whole day. As the MILP-based algorithm is not scalable, we only conducted the experiments with FIFO, FIFO-C, AsQ, FF, and GIO algorithms.

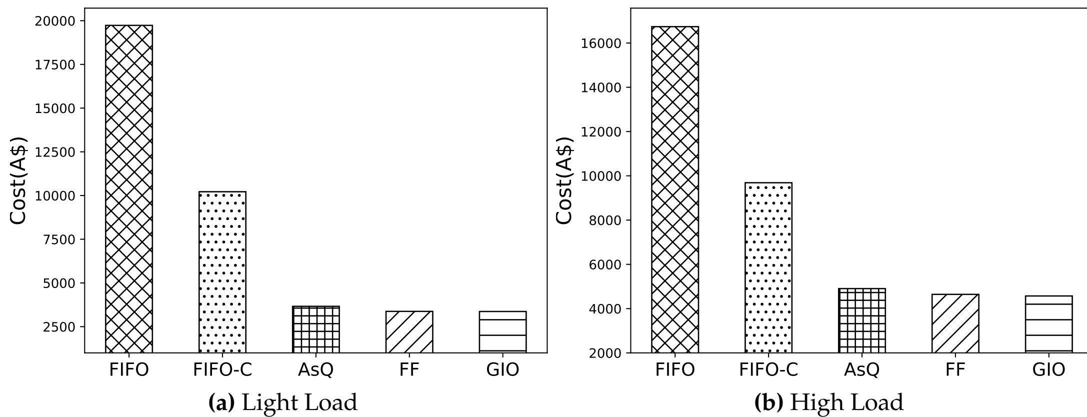


Figure 5.5: Cost comparison in large-scale simulation (the lower the better). FIFO incurs a very high cost as round-robin placements of executors lead to many active VMs simultaneously.

Fig. 5.5 shows the cost comparison results between the scheduling algorithms in both light load and high load scenarios for the large-scale experiment. It can be seen that both FF and GIO outperform the default FIFO and FIFO-C by a significant margin and reduce the cost up to 80%. The AsQ algorithm also tries to find cost-efficient placements in local-only or cloud-only settings. However, as our approaches leverage the hybrid setting to squeeze out spare resources in all the VMs across the cluster, the FF and GIO algorithms reduce the cost up to 15% as compared to the AsQ. Note that, for the small-scale setup, AsQ algorithm performed poorly as compared to the FIFO-C, this is due to the fact that there is limited resource availability in a small cluster, so local-only or cloud-only mode of placement is heavily punished in a higher load. However, for the large-scale setup,

both the local and cloud portion of the cluster have sufficient resources, thus the AsQ outperforms the FIFO-C.

Table 5.5: Average Scheduling Delay (large-scale)

Algorithm	Average Scheduling Delay
FIFO	0.20 μs
FIFO-C	0.24 μs
AsQ	0.0.47 μs
FirstFit	0.33 μs
GIO	0.83 μs

Table 5.5 presents the average scheduling delay for all the algorithms. It can be observed that even for a large-scale setup with many jobs, all the algorithms have a scheduling overhead in μs level thus making all of them extremely scalable.

5.7 Performance Evaluation - Real Experiments

To show the applicability of the proposed algorithms in a real scenario and to validate the results from the simulation experiments, we have conducted real experiments on a Mesos cluster. This section presents the implemented system, experimental setup, benchmark applications and experimental results regarding different aspects of job scheduling.

5.7.1 System Implementation

We have developed a prototype system to evaluate the performance of the proposed job scheduling algorithms in a real hybrid cloud setup. Fig. 5.6 shows the architecture of the system. To implement any scheduling policy, the capability of placing an executor in any VM is needed. Apache Mesos [10] cluster manager provides this functionality by dynamic resource reservations, where any type of resource (e.g., CPU cores or memory)

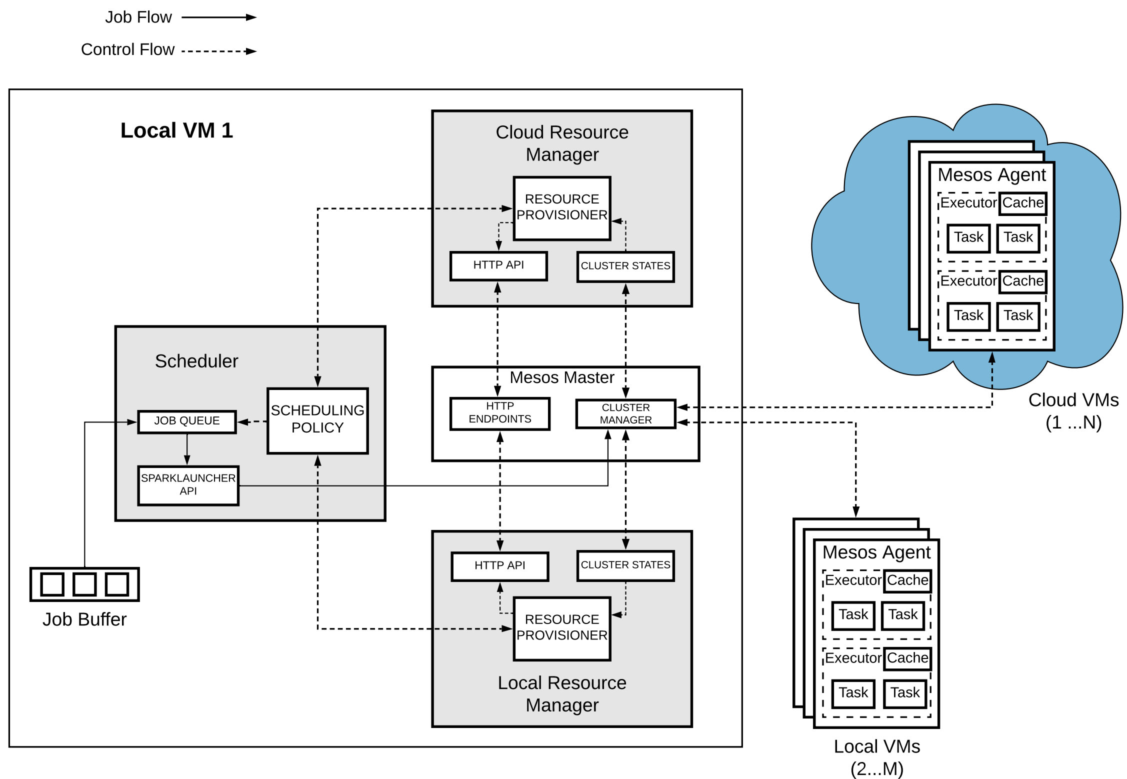


Figure 5.6: System Architecture. The resource managers communicate with the Mesos cluster manager through the REST APIs. The Mesos master is deployed in the local part of the cluster (local VM-1).

can be reserved in any VM so that only the desired executor can run with the reserved resources. Mesos provides HTTP APIs to control dynamic resource reservation of a cluster. Therefore, a scheduler can dynamically place executors in any VM during the scheduling process. As we have a hybrid cluster comprising of both local and cloud VMs, a Mesos cluster can be set up using these VMs, where each VM works as a Mesos agent. Here, each Spark executor runs inside a Mesos container in a Mesos agent.

As shown in the system architecture, we have implemented three additional modules (grey boxes) that work in collaboration with the Mesos master. All these modules are deployed into a local VM along with the Mesos master, which works as a central point of control for both the local and cloud VMs. Thus, from a job's perspective, there is a single cluster. However, the local and cloud VMs are deployed in different regions to

http://mesos.apache.org/documentation/latest/operator-http-api/

exhibit a true hybrid cloud setup. There are two resource managers in the implemented system - Cloud and Local; for managing the VMs. Each resource manager can communicate with the Mesos master using the HTTP APIs for performing resource provisioning. Furthermore, resource managers can fetch cluster states (e.g., job and resource status) from the Mesos master. The scheduling module controls the resource manager modules to perform resource provisioning for any executor. Moreover, it can also instruct the resource managers to turn on/off any VM. When the resources are reserved for all the executors of a job, the scheduling module can directly launch a Spark job in the cluster by using the SparkLauncher API. The developed modules are not extended from the default Spark's framework scheduler. Therefore, it is pluggable to the Mesos cluster manager and can be extended to work with any other Mesos-supported big data frameworks. We have implemented our proposed and baseline scheduling algorithms in the scheduler module. Java programming language was used to implement the proposed modules and scheduling algorithms. OpenStack Boto API was used to automate the VM turn on/off mechanisms.

5.7.2 Real Experiment Setup

We have used Nectar Cloud, a national cloud computing infrastructure for research in Australia to deploy a Mesos cluster. It is a cluster consisting of three different types of VM instances. The detailed VM configurations and quantity used from each type is the same as the small-scale setup shown in Table 5.2. However, the pricing model is different from the simulation pricing models. As shown in Table 5.3 (Pricing Model (Real)), the pricing of the real cloud instances is similar to the VM instance pricing in Amazon AWS (Sydney, Australia). Also, the price of a locally deployed instance is set to be half of the same instance price deployed in cloud. We set up a true hybrid cluster by deploying the VMs in two different regions: Melbourne and Tasmania. We have used the VMs deployed in Melbourne as the local VMs, and the VMs deployed in Tasmania as the cloud VMs. The end-to-end delay between VMs within the same regional boundary

https://spark.apache.org/docs/2.3.0/api/java/index.html?org/apache/spark/launcher/package-summary.html
https://pypi.org/project/boto/
https://nectar.org.au/research-cloud/

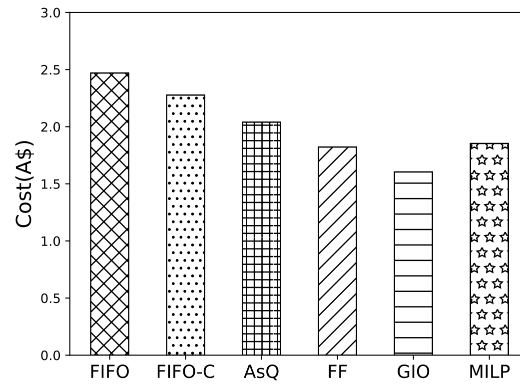
is approximately 10ms, whereas, the end-to-end delay between VMs from different regional boundaries is approximately 40ms. In addition, we have performed *iperf* testing to measure the bandwidth between the VMs. Within the same regional boundary, the bandwidth between the VMs is approximately 2Gbps, whereas, the bandwidth between two VMs from different regional boundaries is around 600Mbps.

Our experimental cluster has 10 VMs with a total of 76 CPU (cores) and 304GB of memory. In each VM, we have installed Apache Mesos (version 1.4.0) and Apache Spark (version 2.3.1). One *m1.large* type VM instance was used as the Mesos master while all the remaining VMs were used as Mesos Agents. The Mesos master node is deployed locally in the Melbourne region. The implemented scheduler and resource manager modules were plugged into the Mesos master node. The developed pluggable modules and the scheduling algorithms are open source and can be used to implement and test new scheduling policies.

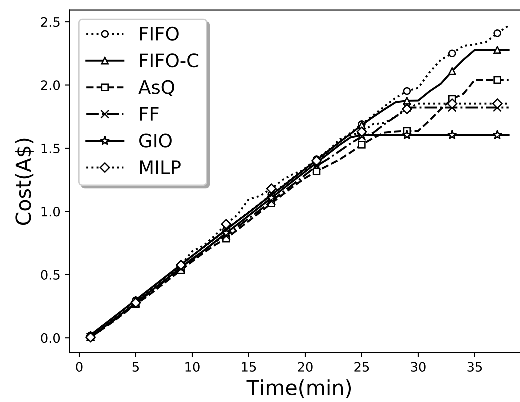
5.7.3 Benchmarking Applications

We have used BigDataBench [67] benchmarking suite for the real experiments. We have taken three types of applications from this benchmark, which are: WordCount (compute-intensive), Sort (memory-intensive), and PageRank (network-intensive). We have randomly mixed all these three applications mentioned above to generate the workload. The job arrival times from the Facebook Hadoop workload trace is extracted for an hour. Collecting job profiles to estimate the completion times is a well-known mechanism. In our experiments, each job is profiled in the real cluster for 10 times, and the average job completion time is taken to use as the estimated job completion time (T_C). These estimated job completion times are used in the problem model by the proposed scheduling algorithms to make scheduling decisions. However, to determine the schedulers performance regarding cost optimization in the real experiment, we measure both the job completion time and the use of VM resources in real-time during the scheduling process for rigorous performance evaluation. The active time remaining for either a cloud or local VM (Δt_j^L for local and Δt_j^C for cloud) can be calculated by using

https://github.com/tawfiqul-islam/Hybrid-Cloud-Scheduler
https://github.com/SWIMProjectUCB/SWIM/wiki/Workloads-repository



(a) Total Cost



(b) Cumulative Cost

Figure 5.7: Cost Comparison between different scheduling algorithms (the lower the better). (a) shows the total cost incurred over a scheduling period, (b) shows the cumulative cost incurred over time.

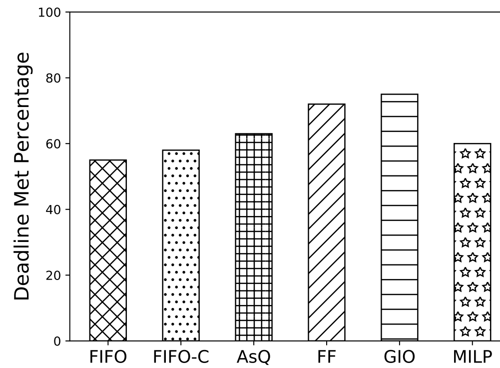


Figure 5.8: Comparison of deadline met percentage between the scheduling algorithms (the higher the better).

the job completion time estimates (T_C) for the jobs which have one or more executors placed in a particular VM. The maximum estimated completion time is taken among these jobs and is subtracted from the current clock time to get an estimate on a VMs active remaining time.

5.7.4 Real Experiment Results

We have evaluated the proposed algorithm regarding cost efficiency, job deadline, and average job completion time. For these experiments, we have used ***Pricing Model (Real)*** as shown in Table 5.3 for the VM pricing, which is similar to the Amazon AWS pricing scheme for the cloud instances. The price of the same instance type deployed locally is considered to be half of the cloud instance price.

Evaluation of Cost Efficiency

In this evaluation, we show the cost efficiency of different scheduling algorithms in the real experimental setup. Both the total cost and the cumulative cost is collected while running 100 jobs (mix of WordCount, Sort, and PageRank) for one hour. Fig. 5.7a exhibits the total cost incurred and Fig. 5.7b shows the cumulative cost incurred by different scheduling algorithms. It can be observed that the default FIFO and FIFO-C algorithms have the highest VM usage cost which increases linearly over time. However, the MILP and the proposed FF and GIO algorithms reduce the cost significantly as they

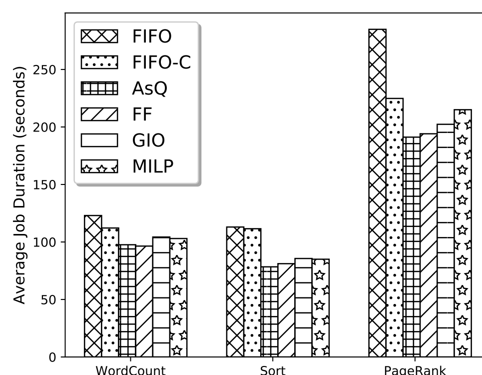


Figure 5.9: Comparison of average job duration between the scheduling algorithms for different types of jobs (the lower the better).

utilize the pricing model of VMs and uses the cheaper VMs for executor placement. Although the AsQ algorithm utilizes the pricing model, it restricts executor placements to local or cloud only. Thus, in a peak load where the cluster does not have sufficient resources, AsQ algorithm fails to utilize the spare resources in VMs by avoiding hybrid placement. The MILP algorithm finds the most cost-optimal placement of executors for each job, due to the scheduling overhead of MILP (computational complexity in some cases), the GIO algorithm performs slightly better and provides a lower cost. Furthermore, the MILP algorithm is only applicable to a small cluster as it is not scalable due to the exponential increase in decision making for a large cluster.

Evaluation of Job Deadline

In this evaluation, we compare the deadline met percentage from different scheduling algorithms. As the FIFO and FIFO-c algorithms do not consider the EDF strategy, they have higher deadline violations as compared to the other algorithms. Although AsQ gives a better deadline met percentage than the default algorithms, it shows a lower deadline met percentage in a peak load, as jobs have to wait longer for a local-only or a cloud-only placement. The proposed FF and GIO algorithms show a higher deadline met percentage due to tight packing of executors and utilizing spare resources in the hybrid setting. Although hybrid placement increases job duration, the jobs do not have to wait longer as the algorithms schedule the jobs as soon as the combined resource

(in both local and cloud VMs) are sufficient to place all the executors. MILP algorithm solves the executor placement in the most cost-efficient way. However, in many cases, it takes a lot of time to find a solution (high scheduling delay), which causes jobs to wait longer and violate deadlines.

Effects on Job Performance

Although it is possible to minimize the cost of using a hybrid cluster by packing more executors in fewer nodes, it causes some performance overhead for CPU/memory-bound jobs. However, when the executors from the same job are distributed over multiple regional VMs, the job completion time increases due to network latency and data transmission delays. As shown in Fig. 5.9, the default FIFO and FIFO-C algorithms always distribute the executors, so most of the placements are hybrid which causes a high average job duration. Network-bound jobs (PageRank) suffer the most, where a lot of network communications take place. The AsQ algorithm provides the lowest average job duration for different types of jobs, as the data transmissions between executors only occur within the same regional boundary. Although the FF, GIO, and MILP algorithms utilize hybrid placement to reduce cost, they have a slightly higher average job duration than the AsQ algorithm. However, due to the tight packing of executors, sometimes these algorithms also place executors in a single region, thus the performance overhead is not as extreme as the FIFO and FIFO-C. Nevertheless, this slight performance degradation is negligible as compared to the cost-saving in the hybrid cluster.

5.8 Summary

In this chapter, we have formulated the SLA-based Spark job scheduling problem in a hybrid cloud as an optimization problem. We have proposed two greedy heuristics-based algorithms to solve the scheduling problem. Besides, we have implemented the proposed algorithms on top of Apache Mesos to show the applicability in real environments. We have compared the proposed approaches in both simulated and real experiments to show the superiority of them over the baseline approaches. The results show

that our proposed algorithms can significantly reduce VM usage costs in a hybrid cloud. Although there are performance overheads due to data transmission delays caused by hybrid placements, it is negligible as compared to the cost-saving benefits. Moreover, the proposed approaches are highly scalable and have low scheduling overhead, which is similar to the native Spark schedulers.

This chapter focuses more on the user's perspective, and when a user submits a Spark job, they do not provide network or disk as resource constraints. Thus, we work on a higher level where we consider resource capacity/demand constraints which are required at the executor creation stage. However, we try to capture the network transmission issues by considering the job duration increase in the problem model. In future, we plan to investigate more on the performance impacts caused by hybrid placements. In addition, we plan to investigate the trade-offs between cost-efficiency and job performance. A more sophisticated model needs to be devised, which can consider both objectives together to generate efficient job schedules. In addition, we would like to explore deeper into the effects of VM turn on/off mechanisms on job performance and cost-efficiency. As Fog computing and Edge computing are becoming increasingly popular, we plan to extend the scheduling algorithms to work with a multi-tier Fog-Edge-Cloud deployed cluster.

Chapter 6

Learning Scheduling Algorithms with Deep Reinforcement Learning (DRL)

Many organizations are shifting towards a cloud deployment of their big data computing clusters. However, job scheduling is a complex problem in the presence of various Service Level Agreement (SLA) objectives such as monetary cost reduction, and job performance improvement. Most of the existing research does not address multiple objectives together and fail to capture the inherent cluster and workload characteristics of a Spark cluster. In this chapter, we formulate the job scheduling problem of a cloud-deployed Spark cluster and propose a novel Reinforcement Learning (RL) model to accommodate the SLA objectives mentioned earlier. Besides, we develop the RL cluster environment on top of TensorFlow (TF) and implement two Deep Reinforce Learning (DRL) based schedulers in TF-Agents framework. The proposed DRL-based scheduling agents work at a fine-grained level to place the executors of jobs while leveraging from the pricing model of cloud VM instances.

6.1 Introduction

BIG data processing frameworks such as Hadoop [12], Spark [6], Storm became extremely popular due to their use in the data analytics domain in many significant areas such as science, business, and research. These frameworks can be deployed in both on-premise physical resources or on the cloud. However, cloud service providers (CSPs) offer flexible, scalable, and affordable computing resources on a pay-as-you-go

https://storm.apache.org/
This chapter is derived from:

- **Muhammed Tawfiqul Islam**, Shanika Karunasekera, and Rajkumar Buyya, "Cost and Performance-oriented Spark Job Scheduling in Cloud with Deep Reinforcement Learning", *IEEE Transactions on Parallel and Distributed Systems (TPDS)* [Under Review].

model. Furthermore, cloud resources are easy to manage and deploy than physical resources. Thus, many organizations are moving towards the deployment of big data analytics cluster on the cloud to avoid the hassle of managing physical resources. In the job scheduling problem of a big data computing cluster, the most important objective is the performance improvement of the jobs. However, when the cluster is deployed on the cloud, job scheduling becomes more complicated in the presence of other crucial Service Level Agreement (SLA) objectives such as the monetary cost reduction.

In this work, we focus on the SLA-based job scheduling problem for a cloud-deployed Apache Spark cluster. We have chosen Apache Spark as it is one of the most prominent frameworks for big data processing. Spark stores intermediate results in memory to speed up processing. Moreover, it is more scalable than other platforms and suitable for running a variety of complex analytics jobs. Spark programs can be implemented in many high-level programming languages, and it also supports different data sources such as HDFS [11], Hbase [14], Cassandra [15], Amazon S3. The data abstraction of Spark is called Resilient Distributed Dataset (RDD) [16], which by design is fault-tolerant.

When a Spark cluster is deployed, it can be used to run one or more jobs. Generally, when a job is submitted for execution, the framework scheduler is responsible for allocating chunks of resources (e.g., CPU, memory), which are called executors. A job can run one or more tasks in parallel with these executors. The default Spark scheduler can create the executors of a job in a distributed fashion in the worker nodes. This approach allows balanced use of the cluster and results in performance benefits to the compute-intensive workloads as interference between co-located executors are avoided. Also, the executors of the jobs can be packed in fewer nodes. Although packed placement puts more stress on the worker nodes, it can improve the performance of the network-intensive jobs as communication between the executors from the same job becomes intra-node. However, depending on the target objective, the users have to determine what type of executor placement should the scheduler use, which often requires inherent knowledge on both the resources and the workload characteristics. Besides, addressing additional SLA objectives such as cost and performance is not possible by the framework

https://aws.amazon.com/s3/

scheduler. There are a lot of existing works [27, 71, 50, 72, 21, 34, 84, 94, 95, 96, 97, 98] that focused on various SLA objectives. However, these works do not consider the implication of executor creations along with the target objectives. Most of the works also assume the cluster setup to be homogeneous. However, this is not the case for a cloud-deployed cluster, where different sizes of the VM instances can be used to deploy the cluster to leverage from the pricing model to reduce the monetary cost. Finally, heuristic-based and performance model-based solutions often focus on a specific objective, and can not be generalized to adapt to a wide range of objectives while considering the inherent characteristics of the workloads.

Recently, Deep Reinforcement Learning (DRL) based approaches are used to solve complex real-world problems [99], where a DRL agent does not have any prior knowledge of the environment. Instead, it interacts with the real environment, explores different situations, and gathers rewards based on its actions. These experiences are used by the agent to build a policy which maximizes the overall reward. The reward is nothing but a model of the desired objectives. In this chapter, we propose DRL-based job scheduling agents for Apache Spark, which addresses the challenges mentioned above. We formulate the scheduling problem and propose an RL model for the job scheduling problem. We also formulate the reward in a way that it can reflect the target SLA objectives such as monetary cost and average job duration reductions. We implement a Q Learning-based agent (Deep Q Learning or DQN), and a policy gradient-based agent (REINFORCE), which automatically learn to schedule jobs efficiently while considering different SLA objectives and the inherent features of the workloads. We also develop a scheduling environment for the cloud-deployed Spark cluster to train the DRL agents. Both the scheduling environment and the agents are developed on top of TensorFlow (TF) Agents. The scheduling agents can interact with the environment to learn about the basics of scheduling, such as satisfying resource capacity and demand constraints for the jobs. Besides, we also train the agents to minimize the monetary cost of VM usage and improve the average job duration of jobs. The environment states and reward signals drive the learning for an agent. When the agent interacts with the scheduling environment, it gets a reward depending on the chosen action. In our proposed RL model for the scheduling problem, an action is a selection of a worker node (VM) for the

creation of an executor of a specific job.

In summary, the **contributions** of this work are as follows:

- We provide an RL model of the Spark job scheduling problem in cloud computing environments. We also formulate the rewards to train DRL-based agents to satisfy resource constraints, optimize cost-efficiency, and reduce average job duration of a cluster.
- We develop a prototype of the RL model in a python environment and plug it to the TF-Agents framework.
- We implement two DeepRL-based agents, DQN and REINFORCE, and train them as scheduling agents in the TF-agent framework.
- We conduct extensive experiments with real-world workload traces to evaluate the performance of the DRL-based scheduling agents and compare them with the baseline schedulers.

The rest of the chapter is organized as follows. In section 6.2, we discuss the existing works related to this chapter. In section 6.3, we formulate the scheduling problem. In section 6.4, we show the proposed RL model. In section 6.5, we describe the proposed DRL-based scheduling agents. In section 6.6, we exhibit the implemented RL environment. In section 6.7, we provide the experimental setup, baseline algorithms, and the performance evaluation of the DRL-based agents. In section 6.7.7, we discuss different strategies learned by the DRL agents and their limitations. Section 6.8 concludes the chapter and highlights future work.

6.2 Related Work

6.2.1 Framework Schedulers

Apache Spark uses the (First in First out) FIFO scheduler by default, which places the executors of a job in a distributed manner (spreads out) to reduce overheads on single worker nodes (or VMs if cloud deployment is considered). Although this strategy

can improve the performance of compute-intensive workloads, due to the increasing network shuffle operations, network-intensive workloads can suffer from performance overheads. Spark can also consolidate the core usage to minimize the total nodes used in the cluster. However, it does not consider the cost of VMs and the runtime of jobs. Therefore, costly VMs might be used for a longer period, incurring a higher VM cost. Fair and DRF [17] based schedulers improve the fairness among multiple jobs in a cluster. However, these schedulers do not improve SLA-objectives such as cost-efficiency in a cloud-deployed cluster.

6.2.2 Performance model and Heuristic-based Schedulers

There are a few works which tried to improve different aspects of scheduling for Spark-based jobs. Most of these approaches build performance models based on different workload and resource characteristics. Then the performance models are used for resource demand prediction, or to design sophisticated heuristics to achieve one or more objectives.

Sparrow [27] is a decentralized scheduler which uses a random sampling-based approach to improve the performance of the default Spark scheduling. Quasar [71] is a cluster manager that minimizes resource utilization of a cluster while satisfying user-supplied application performance targets. It uses collaborative filtering to find the impacts of different resources on an application's performance. Then this information is used for efficient resource allocation and scheduling. Morpheus [50] estimates job performance from historical traces, then performs a packed placement of containers to minimize cluster resource usage cost. Moreover, Morpheus can also re-provision failed jobs dynamically to increase overall cluster performance. Justice [72] uses deadline constraints of each job with historical job execution traces for admission control and resource allocation. It also automatically adapts to workload variations to provide sufficient resources to each job so that the deadline is met. OptEx [21] models the performance of Spark jobs from the profiling information. Then the performance model is used to compose a cost-efficient cluster while deploying each job only with the minimal

https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/FairScheduler.html

set of VMs required to satisfy its deadline. Furthermore, it is assumed that each job has the same executor size, which is the total resource capacity of a VM. Maroulis et al. [34] utilize the DVFS technique to tune the CPU frequencies for the incoming workloads to decrease energy consumption. Li et al. [84] also provided an energy-efficient scheduler where the algorithm assumes that each job has an equal executor size, which is equivalent to the total resource capacity of a VM.

The problems with the performance model and heuristic-based approaches are: (1) they only work on specific SLA objectives and can not be generalized to multiple objectives (2) the performance models depend heavily on the past data, which sometimes can be obsolete due to various changes in the cluster environment (3) it is difficult to tune or modify heuristic-based approaches to incorporate workload and cluster changes. Therefore, recently many researchers are focusing on RL-based approaches to tackle the scheduling problem in a more efficient and scalable manner.

6.2.3 DRL-based Schedulers

The application of Deep Reinforcement Learning (DRL) for job scheduling is relatively new. There are a few works which tried to address different SLA objectives of scheduling cloud-based applications.

Liu et al. [85] developed a hierarchical framework for cloud resource allocation while reducing energy consumption and latency degradation. The global tier uses Q-learning for VM resource allocation. In contrast, the local tier uses an LSTM-based workload predictor and a model-free RL based power manager for local servers. Wei et al. [100] proposed a QoS-aware job scheduling algorithm for applications in a cloud deployment. They used DQN with target network and experience replay to improve the stability of the algorithm. The main objective was to improve the average job response time while maximizing VM resource utilization. DeepRM [94] used REINFORCE, a policy gradient DeepRL algorithm for multi-resource packing in cluster scheduling. The main objective was to minimize the average job slowdowns. However, as all the cluster resources are considered as a big chunk of CPU and memory in the state space, the cluster is assumed to be homogeneous. Decima [95] also uses a policy gradient agent and has a

similar objective as DeepRM. Here, both the agent and the environment was designed to tackle the DAG scheduling problems within each job in Spark, while considering interdependent tasks. Li et al. [101] considered an Actor Critic-based algorithm to deal with the processing of unbounded streams of continuous data with high scalability in Apache Storm. The scheduling problem was to assign workloads to particular worker nodes, while the objective was to reduce the average end-to-end tuple processing time. This work also assumes the cluster setup to be homogeneous and does not consider cost-efficiency. DSS [96] is an automated big-data task scheduling approach in cloud computing environments, which combines DRL and LSTM to automatically predict the VMs to which each incoming big data job should be scheduled to improve the performance of big data analytics while reducing the resource execution cost. Harmony [102] is a deep learning-driven ML cluster scheduler that places training jobs in a way that minimizes interference and maximizes average job completion time. It uses an Actor Critic-based algorithm and job-aware action space exploration with experience replay. Besides, it has a reward prediction model, which is trained using historical samples and used for producing reward for unseen placement. Cheng et al. [97] used a DQN-based algorithm for Spark job scheduling in Cloud. The main objective of this work is to optimize the bandwidth resource cost, along with node and link energy consumption minimization. Spear [103] works to minimize the makespan of complex DAG-based jobs while considering both task dependencies and heterogeneous resource demands at the same time. Spear utilizes Monte Carlo Tree Search (MCTS) in task scheduling and trains a DRL model to guide the expansion and roll-out steps in MCTS. Wu et al. [104] proposed an optimal task allocation scheme with a virtual network mapping algorithm based on deep CNN and value-function based Q-learning. Here, tasks are allocated onto proper physical nodes with the objective being the long-term revenue maximization while satisfying the task requirements. Thamsen et al. [98] used a Gradient Bandit method to improve the resource utilization and job throughput of Spark and Flink jobs, where the RL model learns the co-location goodness of different types of jobs on shared resources.

In summary, most of these existing approaches focus mainly on performance improvement. Furthermore, these works also assume that each job/task will be assigned to one VM or worker-node only. Moreover, many works also assume the cluster nodes to

be homogeneous, which may not be the case when the cluster is deployed on the cloud. Thus, these works do not consider a fine-grained level of executor placement in Spark job scheduling. In contrast, our agent can place executors from the same job in different VMs (when needed, to optimize for a specific policy), and guarantees to launch all of the executors of a job on the required resources. In addition, our scheduling agents can handle different sizes of executors of jobs, and different VM instance sizes with a pricing model. Furthermore, our agent can be trained to optimize a single objective such as cost-efficiency or performance improvement. In addition, our agent can also be trained to balance between multiple objectives. Lastly, the proposed scheduling agents can learn the inherent characteristics of the jobs to find the proper placement strategy to improve the target objectives, without any prior information on the job or the cluster.

6.3 Problem Formulation

We consider a Spark cluster set up using cloud Virtual Machines (VM) as the worker nodes. Generally, Cloud Service Providers (CSPs) offer different instance types for VMs where each type varies on resource capacity. For our problem, we assume that any type or a mix of different types of VM instances can be used to deploy the cluster.

In the deployed cluster, one or more jobs can be submitted by the users; and the users specify the resource demands for their submitted jobs. The job specification contains the total number of executors required and the size of all these executors in-terms of CPU and memory. A job can be of different types and can be submitted at any time. Therefore, job arrival times are stochastic, and the job scheduler in the cluster has no prior knowledge about the arrival of jobs. The scheduler processes each job on an FCFS (First Come First Serve) basis, which is the usual way of handling jobs in a big data cluster. However, the scheduler has to choose the VMs where the executors of the current job should be created. The target of the scheduler is to reduce the overall monetary cost of the whole cluster for all the jobs. In addition, it has an additional target of reducing job completion times. The notation of symbols for the problem formulation can be found in table 6.1.

Suppose N is the total number of VMs that were used to deploy a Spark cluster.

Table 6.1: Definition of Symbols

Symbol	Definition
N	The total number of VMs in the cluster
M	The total number of jobs in the cluster
E	The current total number of executors running in the cluster
ψ	The index set of all the jobs, $\psi = 1, 2, ..., N$
δ	The index set of all the VMs, $\delta = 1, 2, ..., M$
ω	The index set of all the current executors, $\omega = 1, 2, ..., E$
vm^i_{cpu}	CPU capacity of a VM, $i \in \delta$
vm^i_{mem}	Memory capacity of a VM, $i \in \delta$
vm^i_{price}	Unit price of a VM, $i \in \delta$
vm^i_T	The total time a VM was used, $i \in \delta$
e^k_{cpu}	CPU demand of an executor, $k \in \omega$
e^k_{mem}	Memory demand of an executor, $k \in \omega$
job^j_T	The completion time of a job, $j \in \psi$

These VMs can be of any instance types/sizes (which means the resource capacities may vary in-terms of CPU and memory). M is the total number of jobs that needs to be scheduled during the whole scheduling process. When a job is submitted to the cluster, the scheduler has to create the executors in one or more VMs and has to follow the resource capacity constraints of the VMs and the resource demand constraints of the current job. The users submit the resource demand of an executor in two dimensions – CPU cores and memory. Therefore, each executor of a job can be treated as a multi-dimensional box that needs to be placed to a particular VM (bin) in the scheduling process. Therefore, the CPU and memory resource demand and capacity constraints can be defined as follows:

$$\sum_{k \in \omega} (e^k_{cpu} \times x_{ki}) \leq vm^i_{cpu} \qquad \forall i \in \delta \tag{6.1}$$

$$\sum_{k \in \omega} (e^k_{mem} \times x_{ki}) \leq vm^i_{mem} \qquad \forall i \in \delta \tag{6.2}$$

where x_{ki} is a binary decision variable which is set to 1 if the executor k is placed in the VM i; otherwise it is set to 0.

When an executor for a job is created, resources from only 1 VM should be used and the scheduler should not allocate a mix of resource from multiple VMs to one executor. This constraint can be defined as follows:

$$\sum_{i \in \delta} x_{ki} = 1 \qquad \forall k \in \omega \tag{6.3}$$

After the end of the scheduling process, the cost incurred by the scheduler for running the jobs can be defined as follows:

$$Cost_{total} = \sum_{i \in \delta} (vm^i_{price} \times vm^i_T) \tag{6.4}$$

Additionally, we can define the average job completion times for all the jobs as follows:

$$Avg_T = (\sum_{j \in \psi} job^j_T)/M \tag{6.5}$$

As we want to minimize both the cost of using the cluster and the average job com-

pletion time for the jobs, the optimization problem is to minimize the following:

$$\beta \times Cost + (1 - \beta) \times Avg_T \quad (6.6)$$

where $\beta \in [0, 1]$. Here, β is a system parameter which can be set by the user to specify the optimization priority for the scheduler. Note that, equation 6.6 can be generalized to address additional objectives if required.

The above optimization problem is a mixed-integer linear programming (MILP) [90] and non-convex [91], generally known as the NP-hard problem [92]. To solve this problem optimally, an optimal scheduler needs to know the job completion times before making any scheduling decisions. This makes the scheduler design extremely difficult as it requires the collection of job profiles and the modeling of job performance which depends on various system parameters. Furthermore, if the number of jobs, executors and the total cluster size increase, solving the problem optimally may not be feasible. Although, heuristics-based algorithms are highly scalable to solve the problem, they do not generalize over multiple objectives and also do not capture the inherent characteristics of both the cluster and the workload to improve the target goal.

6.4 RL Model

Reinforcement learning (RL) is a general framework where an agent can be trained to complete a task through interacting with an environment. Generally, in RL, the learning algorithm is called the agent, whereas the problem to be solved can be represented as the environment. The agent can continuously interact with the environment and vice versa. During each time step, the agent can take an action on the environment based on its policy ($\pi(a_t|s_t)$). Thus, the action (a_t) of an agent depends on the current state (s_t) of the environment. After taking the action, the agent receives a reward r_{t+1} and the next state (s_{t+1}) from the environment. The main objective of the agent is to improve the policy so that it can maximize the sum of rewards.

In this chapter, the learning agent is a job scheduler which tries to schedule jobs in a Spark cluster while satisfying resource demand constraints of the jobs, and the resource

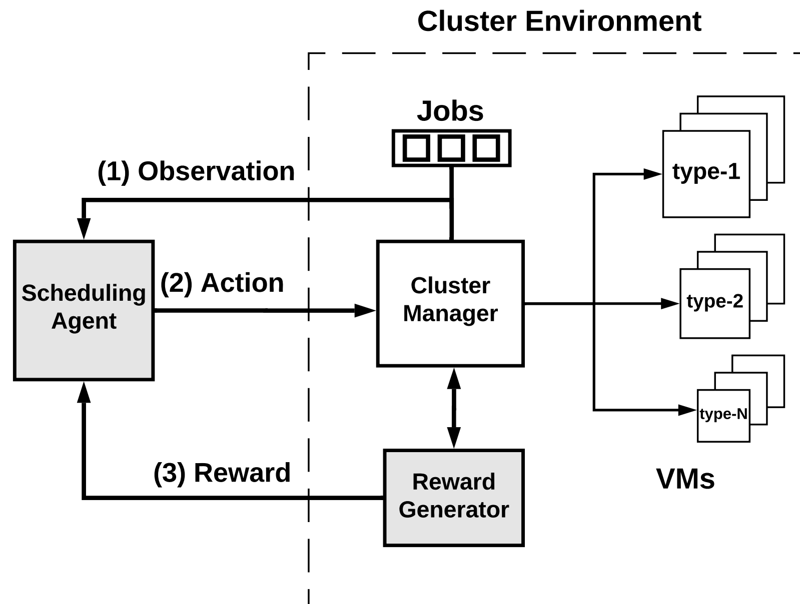


Figure 6.1: The proposed RL model for the job scheduling problem, where a scheduling agent is interacting with the cluster environment.

capacity constraints of the VMs. The reward it gets from the environment is directly associated with the key scheduling objectives such as cost-efficiency, and the reduction of average job duration. Therefore, by maximizing the reward, the agent learns the policy which can optimize the target objectives. Fig. 6.1 shows the proposed RL framework of our job scheduling problem. We treat all the components as part of the cluster environment (highlighted with the big dashed rectangle), except the scheduler. The cluster manager monitors the state of the cluster. It also controls the worker nodes (or cloud VMs) to place executor(s) for any job. In each time-step, the scheduling agent gets an observation from the environment, which includes both the current job's resource requirement, and also the current resource availability of the cluster (exposed by the cluster monitor metrics from the cluster manager). An action is the selection of a specific VM to create a job's executor. When the agent takes an action, it is carried out in the cluster by the cluster manager. After that, the reward generator calculates a reward by evaluating the action on the basis of the predefined target objectives. Note that, in RL environment, the reward given to agent is always external to the agent. However, the RL algorithms can

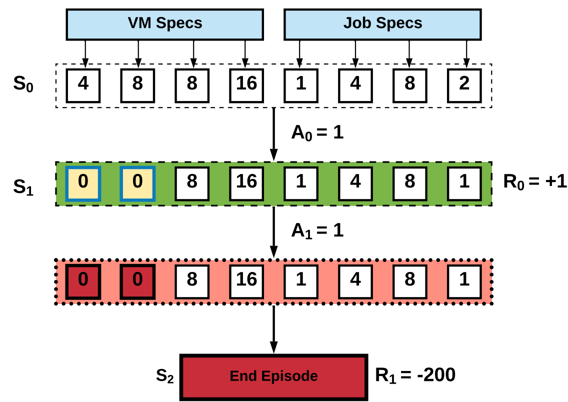
have their internal reward (or parameter) calculations which they continuously update to find a better policy.

We assume the time-steps in our model to be discrete, and event driven. Therefore, the state-space moves from one time-step to the next only after an agent takes an action. The key components of the RL model are specified as follows:

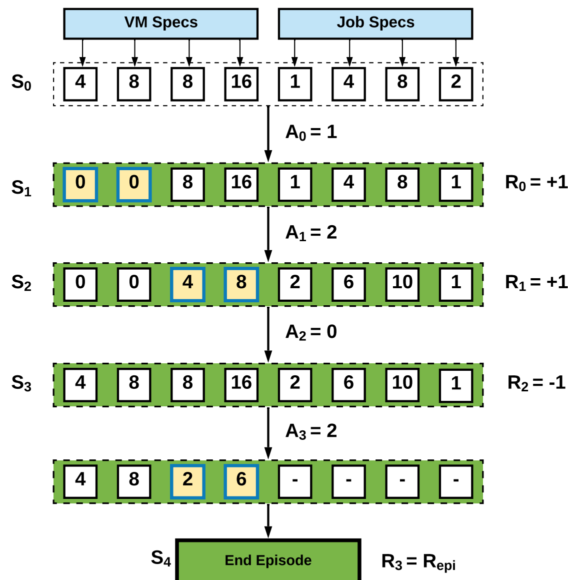
Agent: The agent acts as the scheduler which is responsible to schedule jobs in the cluster. At each time step, it observes the system state and takes an action. Based on the action, it receives a reward and the next observable state from the environment.

Episode: An episode is the time interval from when the agent sees the first job and the cluster state to when it finishes scheduling all the jobs. In addition, for certain bad action taken by the agent can also mean the end of an episode.

State Space: In our scheduling problem, the state is observation an agent gets after taking each action. The scheduling environment is the deployed cluster where the agent can observe the cluster state after taking each action. However, only at the start of the scheduling process or an episode, the agent receives the initial state without taking any action. The cluster states have the following parameters: CPU and memory resource availability of all the VMs in the cluster, the unit price of using each VM in the cluster, and the current job specification which needs to be scheduled. Actions are the decisions or placements the agent (scheduler) makes to allocate resources for each of the executors of a job. Resource allocation for each executor is considered as one action, and after each action, the environment returns the next state to the agent. The current state of the environment can be represented using a 1-dimensional vector, where the first part of the vector is the VM specifications: $[vm^1_{cpu}, vm^1_{mem}, ...vm^N_{cpu}, vm^N_{mem}]$, and the second part of the vector is the current job's specification: $[jobID, e_{cpu}, e_{mem}, j_E]$. Here, $vm^1_{cpu}, ...vm^N_{cpu}$ represents the current CPU availability of all the N VMs of the cluster, whereas $vm^1_{mem}, ...vm^N_{mem}$ represents the current memory availability of all the N VMs of the cluster. $jobID$ represents the current jobID, e_{cpu} and e_{mem} represents the CPU and memory demand of one executor of the current job. As all the executors for one job have the same resource demand, the only other required information is the total number of executors that has to be created for that job, which is represented by j_E. Therefore, the state-space goes larger only with the increase of the size of the cluster (total number of



(a) Failed Episode



(b) Successful Episode

Figure 6.2: Example scenarios for state transitions in the proposed environment.

VMs), and does not depend on the total number of jobs. Each job's specification is only sent to the agent as part of the state after its arrival if all the previous jobs are already scheduled. After each successive action, the cluster resource capacity will be updated due to the executor placement. Therefore, the next state will reflect the updated cluster state after the most recent action. Until all the executors of the current job is placed, the agent will keep receiving the job specification of the current job, with the only change being the j_E parameter which will be reduced by 1 after each executor placement. When it becomes 0 (the job is scheduled successfully), only then the next job specification will be presented along with the updated cluster parameters as the next state.

Action Space: The action is the selection of the VM where one executor for the current job will be created. If the cluster does not have sufficient resources to place one or all the executors of the current job, the agent can also do nothing and wait for previously scheduled jobs to be finished. In addition, to optimize a certain objective (e.g, cost, time), the agent may decide not to schedule a job right after it arrives. Therefore, if there are N number of VMs in the cluster, there are $N + 1$ number of possible discrete actions. Here, we define Action 0 to specify that the agent will be waiting and no executor will be created, where action 1 to N specifies the index of the VM chosen to create an executor for the current job.

Reward: The agent receives a reward whenever it takes an action. There can be either a positive or a negative reward for each action. Positive reward motivates the agent to take a good action and also to optimize the overall reward for the whole episode. In contrast, negative rewards generally train the agent to avoid bad actions. In RL, when we want to maximize the overall reward at the end of each episode, an agent has to consider both the immediate and the discounted future reward to take an action. The overall goal is to maximize the cumulative reward over an episode, so sometimes taking an action which incurs immediate negative reward might be a good step towards a bigger positive rewards in future steps. Now, we define both the immediate reward and episodic reward for an agent.

Suppose, in the worst case, all the VMs are turned on for the whole duration of the episode, where each job took its maximum time to complete because of bad placement decisions. Thus it gives us the maximum cost that can be incurred by a scheduler in an

episode as:

$$Cost_{max} = \sum_{j \in \psi} job^{j}_{Tmax} \times \sum_{i \in \delta} vm^{i}_{price} \tag{6.7}$$

Therefore, if we find the episodic VM usage $Cost_{total}$ incurred by an agent (as shown in Eqn. 6.4), the normalized episodic cost can be defined as:

$$Cost_{normalized} = \frac{Cost_{total}}{Cost_{max}} \tag{6.8}$$

Depending on the priority of the cost objective, the β parameter can be used to find the episodic cost as:

$$Cost_{epi} = \beta \times (1 - Cost_{normalized}) \tag{6.9}$$

In an ideal case where all the jobs' executors are placed according to the job type, we can get the minimum average job completion time for an episode as follows:

$$Avg_{Tmin} = (\sum_{j \in \psi} job^{j}_{Tmin}) / M \tag{6.10}$$

Similarly, if all the jobs' executors are not placed according to the job characteristics, we can get the maximum average job completion time for an episode as follows:

$$Avg_{Tmax} = (\sum_{j \in \psi} job^{j}_{Tmax}) / M \tag{6.11}$$

Therefore, if we find the episodic average job completion time for an agent (as shown in Eqn. 6.5), the normalized episodic average job completion time can be defined as:

$$Avg_{Tnormalized} = \frac{Avg_{T} - Avg_{Tmin}}{Avg_{Tmax} - Avg_{Tmin}} \tag{6.12}$$

Depending on the priority of the average job duration objective, the β parameter can be used to find the episodic average job duration as:

$$Avg_{Tepi} = (1 - \beta) \times (1 - Avg_{Tnormalized}) \tag{6.13}$$

Let R_{fixed} is a fixed episodic reward which will be scaled up or down based on how the agent performs in each episode to maximize the objective function. Thus the final episodic reward R_{epi} can be defined as:

$$R_{epi} = R_{fixed} \times (Cost_{epi} + Avg_{Tepi}) \tag{6.14}$$

Note that, $\beta \in$ [0,1]. In addition, both $Cost_{epi}$ and $Avg_{Tepi} \in$ [0,1]. Therefore, the sum of $Cost_{epi}$ and Avg_{Tepi} can be at most 1, which will lead to a reward of exactly R_{fixed}. For example, if β is chosen to be 0, it means an agent will be trained to reduce average job duration only. In the best case scenario, if an agent can achieve Avg_T to be equal to Avg_{Tmin}, the value of Avg_{Tepi} will be 1 and the value of $Cost_{epi}$ will be 0. Therefore, the value R_{epi} will be equal to R_{fixed} which indicates the agent has learned the most time-optimized policy.

Example workout of the state-action-reward space: We show an example workout of the state, action and reward of the proposed RL model in Fig. 6.2. In this example scheduling scenarios, the cluster is composed with 2 VMs with specifications: $VM_1 \rightarrow \{cpu = 4, mem = 8\}$, and $VM_2 \rightarrow \{cpu = 8, mem = 16\}$. In addition, two jobs arrive one after another with specifications: $job_1 \rightarrow \{jobID = 1, e_{cpu} = 4, e_{mem} = 8, j_E = 2\}$, and $job_2 \rightarrow \{jobID = 2, e_{cpu} = 6, e_{mem} = 10, j_E = 1\}$. Now, Fig. 6.2a shows a scenario where the agent has chosen VM_1 twice to place both of the executors of job_1. After the first placement, the agent received a positive reward $R_0 = 1$ as it was a valid placement. As VM_1 did not have any space left to accommodate anything, the second action taken by the agent was invalid, thus the environment did not execute that action. Instead, the episode was terminated and the agent was given a high negative reward (-200). In the second scenario shown in Fig. 6.2b, the agent successfully placed all the executors for job_1. Then as there was not sufficient resources to place the executor of the job_2, the agent has chosen to wait (Action 0) for resources to be freed. After job_1 is finished, resources are freed. Then the agent successfully placed the executor for job_2, ends the episode and gets the episodic reward.

6.5 DeepRL Agents for Job Scheduling

To solve the job scheduling problem in the proposed RL environment, we use two DRL-based algorithms. The first one is Deep Q Learning (DQN), which is a Q-Learning based approach. The other one is a policy gradient algorithm which is called REINFORCE. Both of the algorithms work with any RL environment with discrete state and action spaces.

6.5.1 DQN Agent

Q-Learning:

Q-Learning works by finding the *Quality* of a state-action value, which is called Q-function. Q-function of a policy π, $Q^{\pi}(s,a)$ measures the expected sum of rewards acquired from state s by taking action a first and then using policy π at each step after that. The optimal Q-function $Q^*(s,a)$ is defined as the maximum return that can be received by an optimal policy. The optimal Q-function can be defined as follows by the Bellman optimality:

$$Q^*(s,a) = \mathbb{E}\left[r + \gamma \max_{a'} Q^*(s',a')\right] \tag{6.15}$$

Here, γ is the discount factor which determines the priority of a future reward. For example, a high value of γ helps the learning agent to achieve more future rewards, while a low value in γ motivates to focus only on the immediate reward. For the optimal policy, the total sum of rewards can be received by following the policy until the end of a successful episode. The expectation is measured over the distribution of immediate rewards of r and the possible next states s'.

In Q-Learning, the Bellman optimality equation is used as an iterative update $Q_{i+1}(s,a) \leftarrow \mathbb{E}\left[r + \gamma \max_{a'} Q_i(s',a')\right]$, and it is proved that it converges to the optimal function Q^*, i.e. $Q_i \rightarrow Q^*$ as $i \rightarrow \infty$ [105].

Deep Q-Learning (DQN):

Q-learning can be solved as dynamic programming (DP) problem, where we can represent the Q-function as a 2-dimensional matrix containing values for each combination of s and a. However, in high-dimensional spaces (the total number of state and action pairs are huge), the tabular Q-learning solution is infeasible. Therefore, a neural is generally trained with parameters θ, to approximate the Q-values, i.e., $Q(s,a;\theta) \approx Q^*(s,a)$. Here, the following loss at each step i needs to be minimized:

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s' \sim \rho(.)} \left[(y_i - Q(s,a;\theta_i))^2 \right] \tag{6.16}$$

where $y_i = r + \gamma \max_{a'} Q(s', a'; \theta_{i-1})$.

Here, ρ is the distribution over transitions $\{s, a, r, s'\}$ sampled from the environment. y_i is called the Temporal Difference (TD) target, and $y_i - Q$ is called the TD error.

Note that the target y_i is a changing target. In supervised learning, we have a fixed target. Therefore, we can train a neural net to keep moving towards the target at each step by reducing the loss. However, in RL, as we keep learning about the environment gradually, the target y_i is always improving, and it seems like a moving target to the network, thus making it unstable. A target network has fixed network parameters as it is a sample from the previous iterations. Thus, the network parameters from the target network are used to update the current network for stable training.

Furthermore, we want our input data to be independent and identically distributed (i.i.d.). However, within the same trajectory (or episode), the iterations are correlated. While in a training iteration, we update model parameters to move $Q(s,a)$ closer to the ground truth. These updates will influence other estimations and will destabilize the network. Therefore, a circular **replay-buffer** can be used to hold the previous transitions (state, action, reward samples) from the environment. Therefore, a mini-batch of samples from the replay buffer is used to train the deep neural network so that the data will be more independent and similar to i.i.d.

DQN is an off-policy algorithm that it uses a different policy while collecting data from the environment. The reason is if the ongoing improved policy is used all the time, the algorithm may diverge to a sub-optimal policy due to the insufficient coverage of the

state-action space. Therefore, an ϵ-greedy policy is used that selects the greedy action with probability $1 - \epsilon$ and a random action with probability ϵ so that it can observe any unexplored states, which ensures that the algorithm does not get stuck in local maxima. The DQN [106] algorithm we use with replay buffer and target network is summarized in Algorithm 8.

Algorithm 8: DQN Algorithm

1 **foreach** *iteration* 1 ... N **do**
2 Collect some samples from the environment by using the collect policy (ϵ-greedy), and store the samples in the replay buffer
3 Sample a batch of data from the replay buffer
4 Update the agent's network parameter θ (Using Eqn. 6.16)

6.5.2 REINFORCE Agent

DQN optimizes for the state-action values, and by doing so, it indirectly optimizes for the policy. However, the policy gradient methods operate on modelling and optimizing the policy directly. The policy is usually modelled with a parameterized function with respect to θ, written as π_θ. Accordingly, $\pi(a_t|s_t)$ is the probability of choosing the action a_t given a state s_t at time step t. The amount of the reward an agent can get depends on this policy.

In a conventional policy gradient algorithm, a batch of samples is collected in each iteration, then the update shown in Eqn. 6.17 is applied to the policy using the collected samples.

$$\nabla_\theta \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \gamma^t r_t\right] = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta log \pi_\theta(a_t|s_t) R_t\right] \tag{6.17}$$

Here, γ is the discount factor, whereas s_t, a_t, and r_t are used to represent the state, action, and reward at time t, respectively. T is the length of any single episode. R_t is the discounted cumulative return, which can be computed as shown in Eqn. 6.18.

$$R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_t \tag{6.18}$$

Here, t' starts from the current time step t, which means that if the current action is taken, we will get an immediate reward of r_t, which also influences on how much reward we can accumulate up to the end of the episode.

The expected return is shown in Eqn. 6.17 uses the maximum log-likelihood, which measures the likelihood of an observed data. In RL context, it means how likely we can expect the current trajectory under the current policy. When the likelihood is multiplied with the reward, the likelihood of a policy is increased if it generates a positive reward. On the other hand, the likelihood of the policy is decreased if it gives a less or a negative reward. In summary, the model tries to keep the policy which worked better and tends to throw away policies which did not work well. However, as the formula is shown as an expectation, it cannot be used directly. Therefore, a sampling-based estimator is used instead, which is shown in Eqn. 6.19.

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta log\pi_\theta(a_{i,t}|s_{i,t})\right)\left(\sum_{t=1}^{T} r(s_{i,t},a_{i,t})\right) \tag{6.19}$$

Algorithm 9: REINFORCE Algorithm

1 **foreach** *iteration* 1 ... N **do**
2 Sample τ_i from $\pi_\theta(a_t|s_t)$ by following the current policy in the environment
3 Find the policy gradient $\nabla_\theta J(\theta)$ (Using Eqn. 6.19)
4 $\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$

Here, $\nabla_\theta J(\theta)$ is the policy gradient of the target objective J, parameterized with θ. We also assume that in each iteration, N trajectories are sampled ($\tau_1, ..., \tau_N$), where each trajectory τ_i is a list of states, actions, and rewards: $\tau_i = s^i_t, a^i_t, r^i_t$ for time-steps $t = 0$ to $t = T_i$. In this work, we use the **REINFORCE** [107] algorithm, as shown in Algorithm 9. This algorithm works by utilizing Monte Carlo roll-outs (learning by computing the reward after executing a whole episode). After the collection step (line 2), the algorithm updates the underlying network using the updated policy gradient with a learning parameter α (line 4). Note that, while sampling a trajectory, the ϵ-greedy policy is used.

6.6 RL Environment Implementation

We have developed a simulation environment in *Python* to represent a cloud-deployed Spark cluster. The environment holds the state of the cluster, and an agent can interact with it by observing states, and taking any action. Whenever an action is taken, the immediate reward can be observed, but the episodic reward can only be observed after the completion of an episode. The episodic reward may be positive or negative depending on whether an episode was completed successfully or terminated early following a bad action taken by the agent. The features of our developed environment are summarized as follows:

1. The environment exposes the state (comprised of the latest cluster resource statistics and the next job), to the agent in each time step.

2. After an action is taken by an agent, the environment can detect valid/invalid placements and assign positive/negative rewards accordingly.

3. Based on the agent's performance in an episode, the environment can award the episodic reward (the environment acts as a reward generator). Therefore, for a simulated cluster with a workload trace, the environment can derive the cost and time values which are required to find the episodic reward.

4. The environment can also vary the job durations from the goodness of an agent's executor placement.

As mentioned before, instead of representing fixed intervals of real-time; the time-steps refer to arbitrary progressive stages of decision-making and acting. We have incorporated TF-agents API calls to return the transition or termination signals after each time step. Fig. 6.3 shows the workflow of the environment during the agent training process. The red and green circles indicate the events which trigger negative and positive rewards, respectively, from the environment. A summary of the *'action leading to the event and reward'* is summarized in Table 6.2. In this table, the serial No. of each reward corresponds to the red/green circle shown in Fig. 6.3. The implemented environment can be used with TF-agents to train one or more DRL agents. Specifically, the

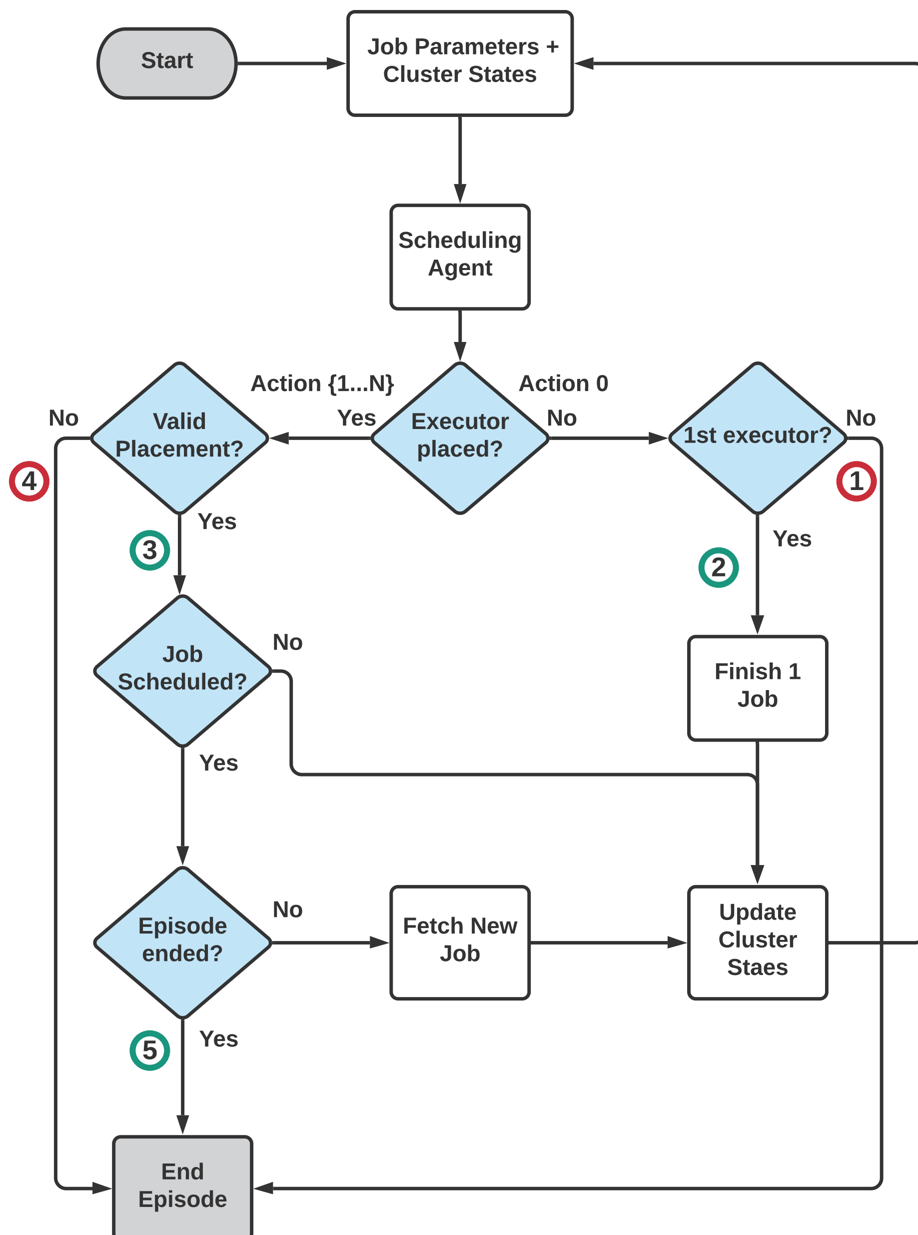


Figure 6.3: The workflow of the proposed environment in response to different agent actions. The red and green circles indicate the events which trigger negative and positive rewards, respectively, from the environment.

Table 6.2: The action-event-reward mapping of the proposed RL environment.

No.	Action	Event	Reward
1	0	Previously placed 1 or more executors of the current job, but now waiting to place any remaining executor (s)	-200
2	0	No placement	-1
3	1 ... N	Proper placement of one executor of the current job	+1
4	1 ... N	Improper placement of an executor: resource capacity or resource demand constraints violation	-200
5	1 ... N	All jobs are scheduled, a successful completion of an episode	R_{epi} (Eqn. 6.14)

agents can be trained to achieve one or more target objectives such as cost-efficiency, performance improvement. As discussed before, we have designed the reward signals to achieve both cost-efficiency and average job duration reduction. The implemented environment can be extended to change or incorporate one or more rewards/objectives. We call the implemented environment **RM_DeepRL**, which is an open-source RL-based cluster scheduling environment with TensorFlow-Agents as the backend.

6.7 Performance Evaluation

In this section, we first discuss the experimental settings which include the cluster resource details, workload generation, and baseline schedulers. Then, we present the evaluation and comparison of the DRL agents with the baseline scheduling algorithms.

6.7.1 Experimental Settings

Cluster Resources: We have chosen different VM instance types with various pricing models so that we can train and evaluate an agent to optimize cost while the cluster is deployed on public cloud. The cluster resource details are summarized in Table 6.3. Note that, the pricing model of the VM instances is similar to the AWS EC2 instance pricing (in Australia).

https://github.com/tawfiqul-islam/RM_DeepRL

Table 6.3: Cluster Resource Details

Instance Type	CPU Cores	Memory (GB)	Quantity	Price
m1.large	4	16	4	$0.24/h
m1.xlarge	8	32	4	$0.48/h
m2.xlarge	12	48	4	$0.72/h

Table 6.4: Hyper-parameters for DRL-agents and the environment parameters.

Parameter	Value	Parameter	Value
R_{fixed}	10000	Optimization Priority (β)	[0.0,0.25,0.50,0.75,1.00]
Batch Size	64	No. of Fully Connected Layers for Q-Network	200
No. of Evaluation Episodes	10	Policy Evaluation Interval	1000
Epsilon (ϵ)	0.001	Training Iteration	10000 (Normal) 20000 (Burst)
Learning Rate (α)	0.001	Optimizer	AdamOptimizer
Discount Factor (γ)	0.9	Job duration increase for a bad placement	30%
Collect Steps per Iteration (DQN) Collect Episodes per Iteration (RE)	10 10	Avg_{Tmin}, Avg_{Tmax}	Profiled from real runs of the corresponding job
Replay Buffer Size	10000	Avg_T, $Cost_{max}$	Dynamically calculated by the environment depending on the cluster and workload specs

Workload: We have used the BigDataBench [67] benchmark suite and took 3 different applications from it as jobs in the cluster which are: WordCount (CPU-intensive), PageRank (Network or IO intensive) and Sort (memory-intensive). We have randomly set job requirements within a range of 1-6 (for CPU cores), 1-10 (for memory in GB), and 1-8 (for total executors) and then profiled each job in the real Spark cluster 10 times to find the average job duration. Note that, the real cluster also has the same cluster resources as mentioned in Table 6.3.

The job arrival times: Job arrival rates of 24 hours is extracted from the Facebook Hadoop Workload Trace to be used as the job arrival times in the simulation. We have

https://github.com/SWIMProjectUCB/SWIM/wiki/Workloads-repository

chosen job arrival patterns: normal (50 jobs arriving in a 1-hour time period), and burst (100 jobs arriving in only 10 minutes).

Baseline Schedulers: We have used 4 different baselines to compare with the DRL-based algorithms. These are:

1. **Round Robin (RR):** The default approach of the Spark Scheduler to distributively place the executors in VMs.

2. **Round Robin Consolidate (RRC):** Another round-robin approach of the Spark scheduler to minimize the total number of VMs used. Note that it works by packing executors on the already running VMs to avoid launching unused VMs.

3. **First Fit (FF):** We develop this baseline to place as many executors as possible to the first available VM to reduce cost.

4. **Integer Linear Programming (ILP):** This algorithm uses a Mixed ILP solver to find optimal placement of all the executos of the current job. During each decision making step, the whole optimization problem is dynamically generated by using the current cluster state and the job specification. In addtion, to improve the performance we have used job profile information to include the estimated job completion time within the model so that the problem can be solved optimally.

   Note that, all the baseline schedulers, and our proposed scheduling agents make dynamic decisions from the current view of the cluster, and do not have a global view of the whole problem. In addition, the schedulers have no knowledge about the job characteristics which determine the placement goodness for a particular type of job.

TensorFlow Cluster details: We have used 4 VMs (each with 16 CPU cores and 64GB of memory) from the Nectar Research Cloud to train the DRL agents. The TensorFlow version 2.0, and TF-Agent version 0.5.0 were installed along with python 3.7 in each of the VMs.

Hyperparameters: Hyper-parameters settings for both DQN and REINFORCE agents, along with other environment parameters are listed in Table 6.4.

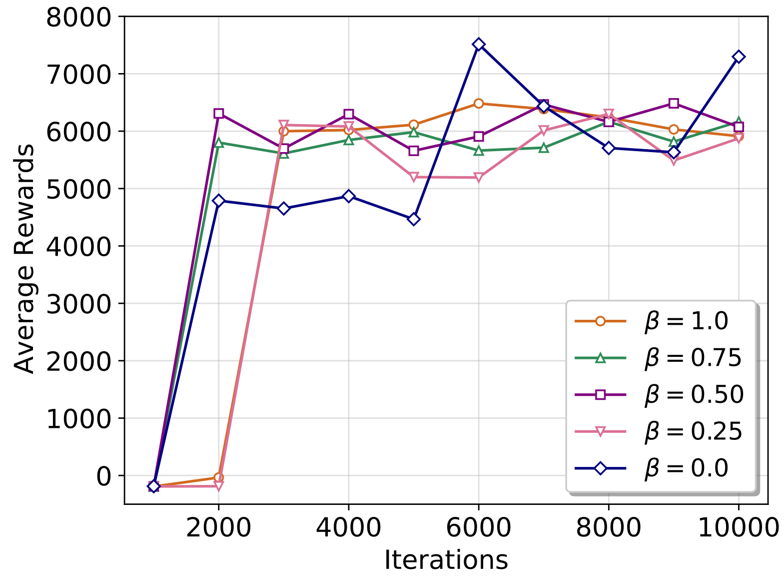
www.nectar.org.au

6.7.2 Convergence of the DRL Agents

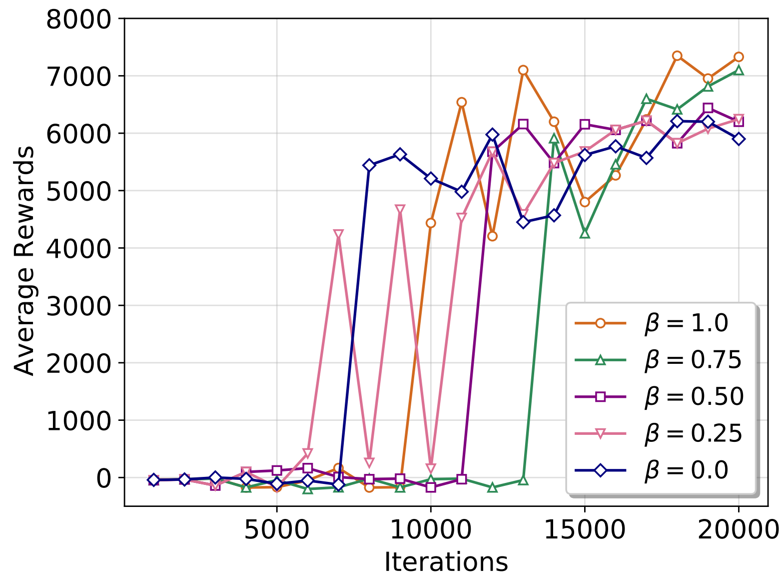
Fig. 6.4 and Fig. 6.5 represent the convergence of the DQN and REINFORCE algorithms, respectively. We have trained the DRL-agents with varying β parameter values to showcase the effects of single or multiple reward maximization. The evaluation of the algorithms is done after every 1000 iterations, where we calculate the average rewards from the 10 test runs of the trained policy. For the normal job arrival pattern, we have trained the agents for 10000 iterations, and for the burst job arrival pattern, we have trained the agents for 20000 iterations.

A higher value of β indicates that the agent is rewarded more for optimizing VM usage cost. In contrast, a lower value of β indicates the agent is optimized more for the reduction of average job duration. We have varied the values of β from 0 to 1, where value 1 indicates that the agent is optimized for cost only. Thus the reward for optimizing average job duration is ignored in the episodic reward. In contrast, a value of 0 of β indicates that the agent is optimized for reducing average job duration only. Any value of β excluding 0 and 1 indicates a mix-mode of operation, where an agent tries to optimize both rewards with different priorities (for values 0.25 and 0.75) or with the same priority (for the value of 0.50). Note that, the episodic rewards can vary and are calculated based on the cluster resource state, job specifications and arrival rates. Additionally, the final episodic reward varies between different optimization targets, so various training settings result in distinctive maximal rewards for an episode.

Fig. 6.4a and 6.4b represent average rewards accumulated by the DQN agent in training for the normal and burst job arrival patterns, respectively. Similarly, Fig. 6.5a and 6.5b represent average reward accumulation in training iterations by the REINFORCE agent. Note that, the average rewards are made up with both fixed rewards received for each successive executor placement and the final episodic reward, and is not the same as the actual VM usage cost or average job duration values. However, accumulating higher total reward implies the agent has learned a better policy which can optimize the actual objectives. Initially, both agents receive negative rewards and gradually start receiving more rewards after exploring the state-space over multiple iterations. Due to the randomness induced by the ϵ-greedy, sometimes the rewards drop for both algorithms. However, the training of REINFORCE agent is more stable than the DQN agent. Both

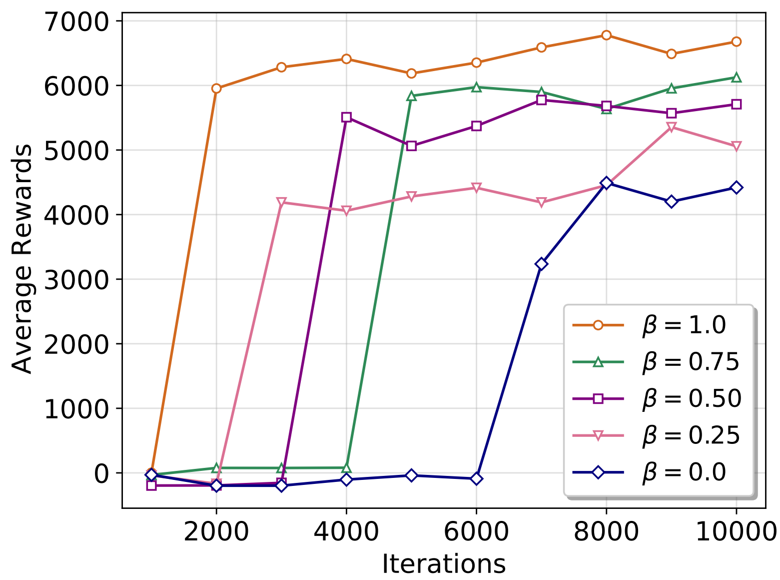


(a) Normal Job Arrival

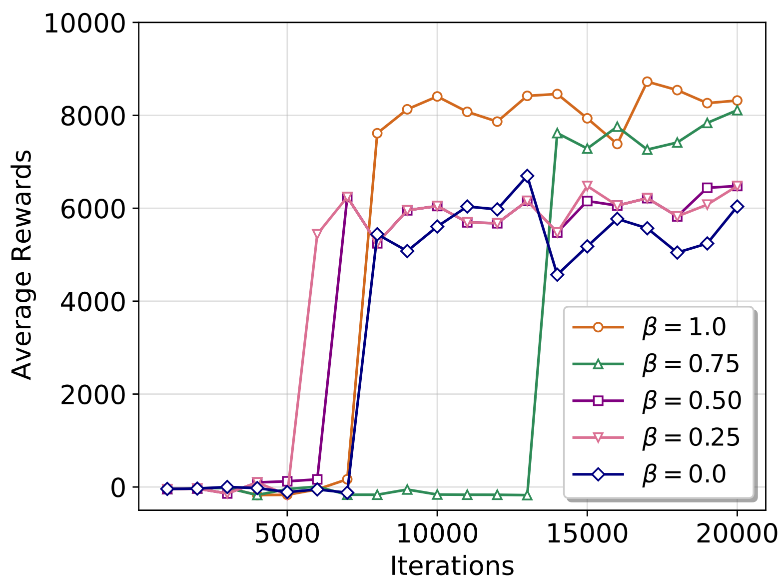


(b) Burst Job Arrival

Figure 6.4: Convergence of the DQN algorithm.



(a) Normal Job Arrival



(b) Burst Job Arrival

Figure 6.5: Convergence of the REINFORCE algorithm.

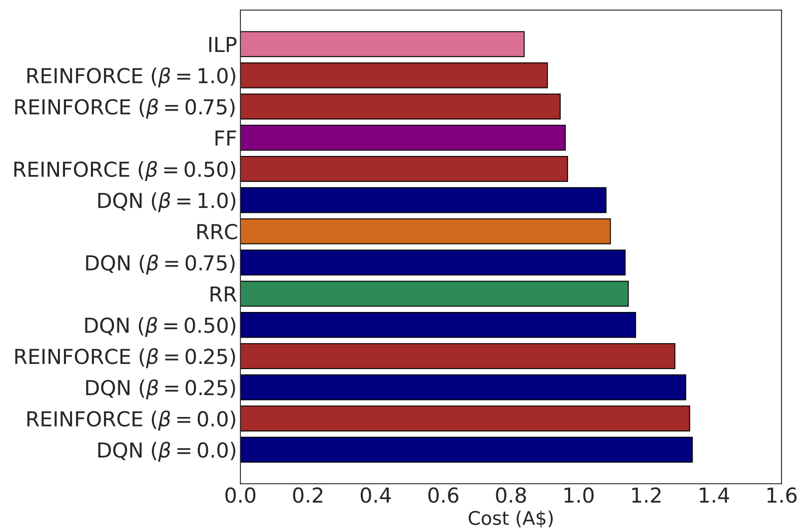
agents required more time to converge with the workload with burst job arrival pattern as there are more jobs (large state-action space), and the agents have to learn to wait (action 0) when the cluster does not have sufficient resources to accommodate the resource requirements of a burst of jobs.

6.7.3 Learning Resource Constraints

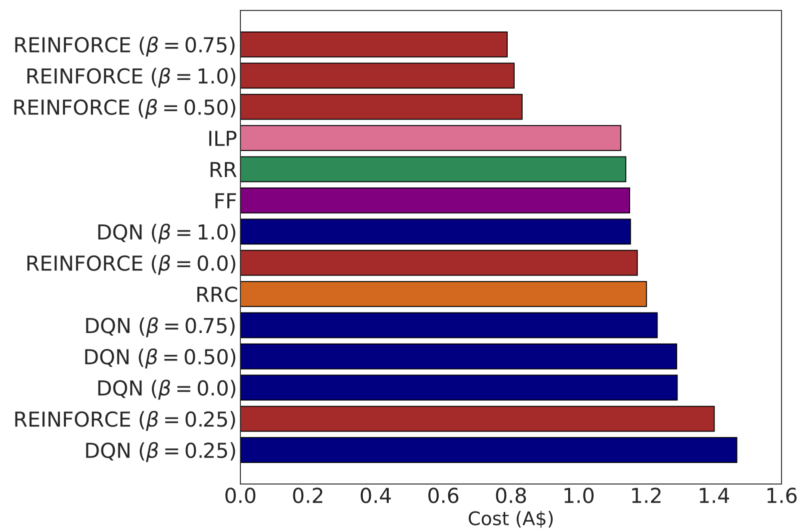
It can be also observed from both Fig. 6.4 and Fig. 6.5 that the training environment works properly to train the agent to avoid bad actions such as violating resource capacity and demand constraints with the use of huge negative rewards. Therefore, at the start of the training process, both the algorithms incur huge negative rewards. However, after taking some good actions (executor placements while satisfying the constraints), the environment awards small immediate rewards, which motivates the agents to eventually complete the episode by scheduling all the jobs successfully. After the agents learn to schedule properly without violating the resource constraints, it can start learning to optimize the target objectives as it can observe different episodic reward depending on all the actions taken over a whole episode.

6.7.4 Evaluation of Cost-efficiency

We evaluate the proposed DRL-based agents and the baseline scheduling algorithms regarding VM usage cost over a whole scheduling episode. In particular, we calculate the total usage time of each VM in the cluster and find the total cost of using the cluster. Fig. 6.6a exhibits the comparison of the scheduling algorithm while minimizing the VM usage cost with a normal job arrival pattern. As the job arrivals are sparse, the algorithms significant job duration increase due to improper placements. Therefore, tight packing on fewer VMs results in lower VM usage cost. The ILP algorithm outperforms all the other algorithms and incurs the lowest VM usage cost (0.84$), as it utilizes the job completion time estimates to find the cost-optimal placements of executors. Both REINFORCE (β=1.0, cost-optimized), and REINFORCE (β=0.75) performs closely to the ILP algorithm, and incur 0.91$ and 0.95$, respectively. Therefore, these agents have only a slight increase in cost from the ILP algorithm, which is 8% and 12%, respectively.



(a) Normal Job Arrival



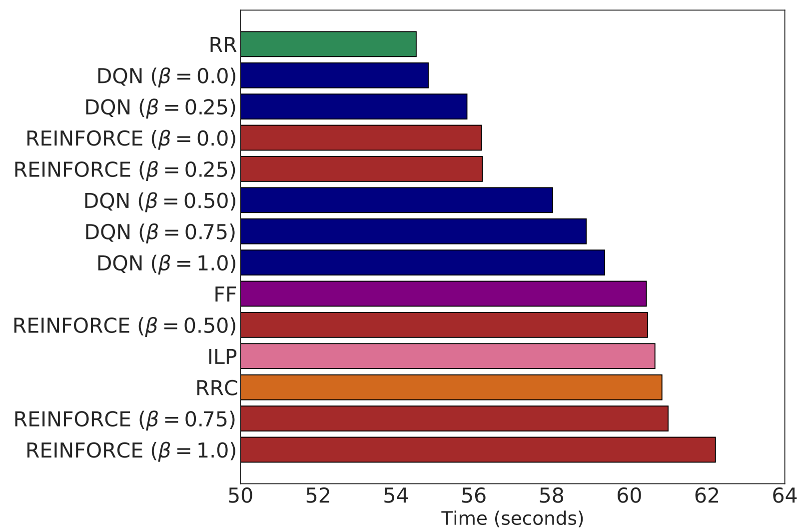
(b) Burst Job Arrival

Figure 6.6: Comparison of the total VM usage cost incurred by different scheduling algorithms in a scheduling episode.

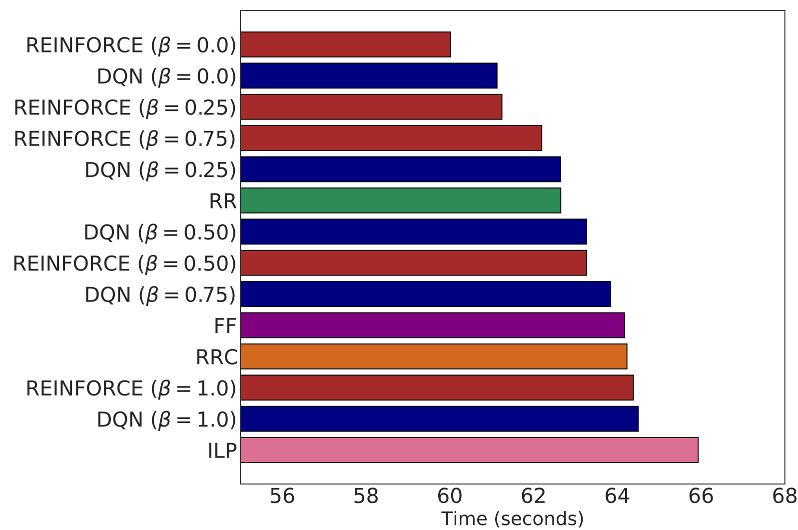
6.7.5 Evaluation of Job Duration

For the burst job arrival pattern, often there are not enough cluster resources to schedule all the jobs, so the scheduling algorithms have to wait until resources are freed up by the already running jobs. In addition, as the job arrival is dense, there can be a lot of job duration increase due to the bad placements for a particular type of job. For example, if a network-bound job is scheduled across multiple VMs, the job run-time will increase, which might lead to an increasing VM usage cost. Although the ILP algorithm utilizes job completion time estimates, it still suffers from job duration increase if the job placement is not matched with the job characteristics, which is reflected in Fig. 6.6b. The REINFORCE agents achieve a significant cost-benefit, where three REINFORCE agents (β values of 0.75, 1.00 and 0.50) incur only 0.78\$, 0.81\$, and 0.83\$, respectively. In comparison, the best baseline ILP incurs 1.12\$, which is 30% more than the best REINFORCE agent (β = 0.75). Although surprising, the REINFORCE agent with β = 0.75 optimizes VM costs better than the β = 1.0 version, because for a burst job arrival, taking both objectives into consideration trains a better policy which in the long-run can optimize cost more effectively. The RR algorithm does not perform well because it cares only about distributing the executors, which results in higher VM usage cost. Although both FF and RRC algorithms try to minimize VM usages, for network-bound jobs, restricting to use only a few VMs can instead increase the cost due to the job duration increase. The DQN agents show a mediocre performance while minimizing VM usage cost, as the trained policy is not as good as the REINFORCE to learn the underlying job characteristics to minimize VM usage time.

We calculate the average job duration for all the jobs scheduled in an episode to compare the performance of the scheduling algorithms. Fig. 6.7a shows the comparison between the scheduling algorithms while reducing the average job duration. For the normal job arrival pattern, the RR algorithm performs the best as it cares only about distributing the jobs among multiple VMs. As there are more memory-bound and CPU-bound jobs combined than the network-bound jobs, the RR algorithm does not acquire significant job duration penalties due to distributed placement of network-bound jobs. RR algorithm is closely followed by the time-optimized versions of the DQN (β=0.0 and β=0.25) and the REINFORCE (β=0.0 and β=0.25) agents, respectively. The DQN

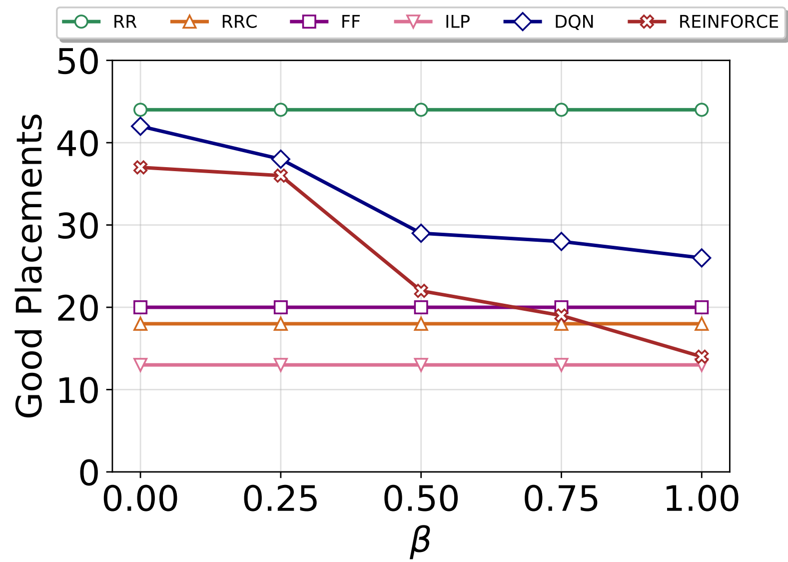


(a) Normal Job Arrival

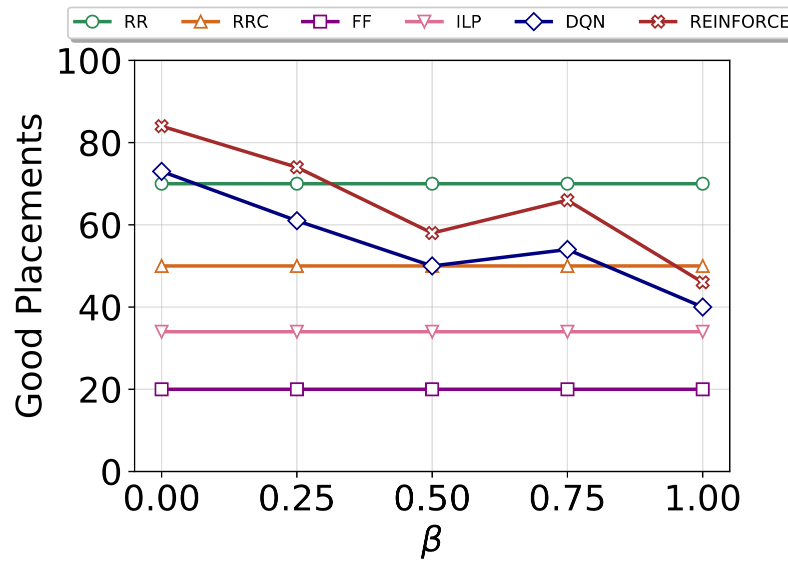


(b) Burst Job Arrival

Figure 6.7: Comparison between the scheduling algorithms regarding the average job duration in a scheduling episode.



(a) Normal Job Arrival



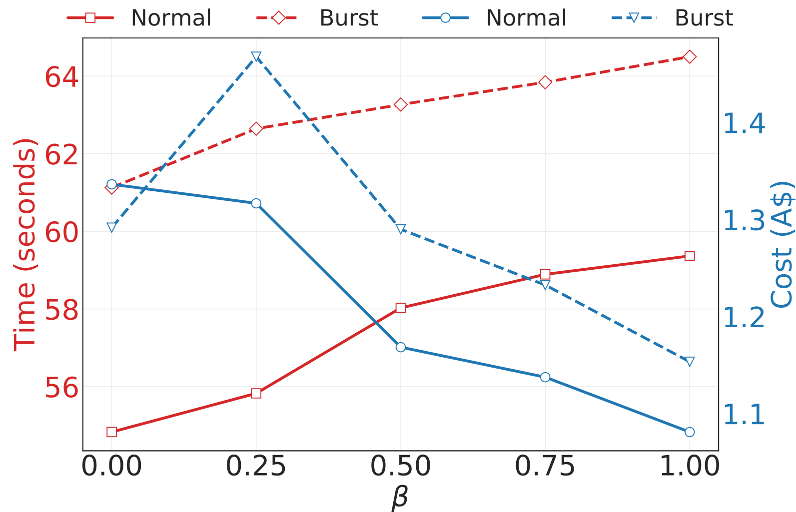
(b) Burst Job Arrival

Figure 6.8: Comparison of good placement decisions made by each scheduling algorithm in a scheduling episode.

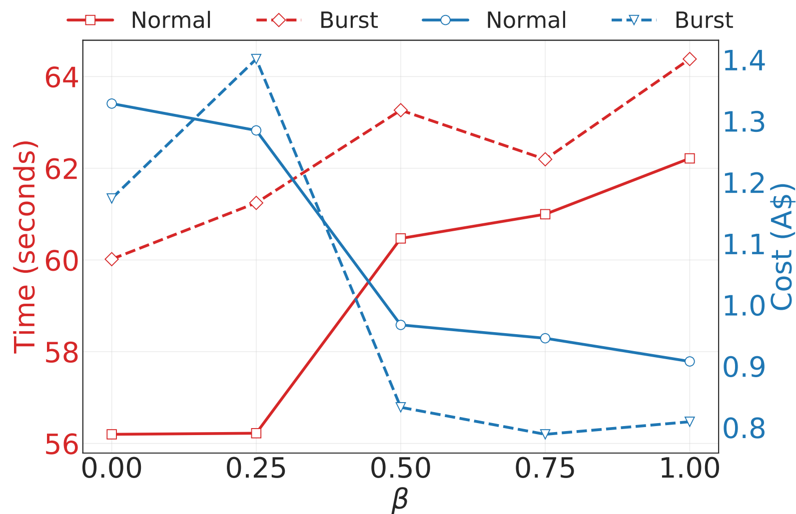
(β=0.0) only increases the average job duration by 1%, whereas the REINFORCE (β=0.25) increases the average job duration by 4% when compared with the RR algorithm.

For the burst job arrival pattern, jobs often have to wait before more resources are available. In addition, if the job placement is not matched with the job characteristics, the completion time of the jobs will increase, which results in a higher average job duration. In this scenario, our proposed REINFORCE and DQN agents can capture the underlying relationship between job duration and job placement goodness and incorporates this information as a strategy to reduce job duration in the trained policy. As shown in Fig. 6.7b, REINFORCE (β=0.0) outperforms the best among the baseline algorithms RR and reduces the average job duration by 2%.

The underlying job characteristics reflect the ideal placement for a particular type of job. In addition, it impacts both the cost-minimization and job duration reduction objectives. Therefore, we also measured the number of good placements by each algorithm. Fig. 6.8a and 6.8b represents the good placement decisions made by all the algorithms in normal and burst job arrival patterns, respectively. As the baseline scheduling algorithms operate on a fixed objective and can not capture workload characteristics, the number of good placement decisions are fixed for each of the baseline algorithms. Therefore, the β parameter does not affect these algorithms, so the results from these algorithms appear as horizontal lines. However, both DQN and REINFORCE agents can be tuned to be cost-optimized, time-optimized or a mix of both. There is a decreasing trend in the number of good placements seen for both agents while the β parameter is increased (while moving towards cost-optimized from time-optimized version). The performance of DQN and REINFORCE discussed for average job duration reduction can be explained from these graphs. It can be observed that the DQN agent makes more good placements than the REINFORCE agent for the normal job arrival pattern, which results in lower average job duration for the DQN agents. In contrast, the REINFORCE agent makes more good placement decisions for the burst job arrival pattern, thus reducing the average job duration better than the DQN agent.



(a) DQN Agent



(b) REINFORCE Agent

Figure 6.9: The effects of the β parameter while using a multi-objective episodic reward in the RL environment. β=0.0 means time optimized only. β=1.0 means cost optimized only. Rest of the values represent a mix mode where both rewards have shared priority.

6.7.6 Evaluation of Multiple Reward Maximization

Fig. 6.9a and 6.9b exhibits the effects of the β parameter while optimizing multiple rewards. The solid lines represent the normal job arrival pattern, whereas the dashed lines represent the burst job arrival pattern. In addition, if a line is in blue colour, it represents the effect on time, whereas a red line reflects the effect on cost. It can be observed that both DQN and REINFORCE agents show stable results while maximizing one or more objectives. With the increase of β value, the agents are trained more towards optimizing the cost instead of the time reduction. While multiple rewards need to be optimized, the β parameter can be tuned to train the agents to learn a balanced policy which prioritizes both objectives. For example, in Fig. 6.9a, the solid blue and red lines at β=0.50 represents a DQN agent which provides a balanced outcome while optimizing both cost and time.

6.7.7 Learned Strategies

Here, we summarize the different strategies learned by the agents:

1. The DRL agents learn the VM capacity and job demand constraints through the negative reward from the environment when taking bad actions such as constraint violation or partial executor placements.

2. The DRL agents learn to optimize cost by packing executors in fewer VMs. However, depending on the job characteristics, they also learn to spread out executors to avoid job duration increase, which in turn results in better cost and time rewards (as showcased in the placement goodness evaluation graphs).

3. The agents can learn to handle both normal or burst job arrival patterns. The DRL agents decide to wait by choosing action 0 continuously when no cluster resources are available to run any more jobs. Although the agents incur a negative immediate reward (-1) while waiting to schedule, they choose it to avoid invalid or partial executor placement, which will lead to high negative rewards.

4. The agents can also learn a stable policy which balances multiple rewards (as shown from the β parameter tuning graphs).

6.8 Summary

Job scheduling for big data applications in the cloud environment is a challenging problem due to the many inherent VM and workload characteristics. Traditional framework schedulers, LP-based optimization, and heuristic-based approaches mainly focus on a particular objective and can not be generalized to optimize multiple objectives while capturing or learning the underlying resource or workload characteristics. In this chapter, we have introduced an RL model for the problem of Spark job scheduling in the cloud environment. We have developed a prototype RL environment in TF-agents which can be utilized to train DRL-based agents to optimize one or multiple objectives. In addition, we have used our prototype RL environment to train two DRL-based agents, namely DQN and REINFORCE. We have designed sophisticated reward signals which help the DRL agents to learn resource constraints, job performance variability, and cluster VM usage cost. The agents can learn to optimize the target objectives without any prior information about the jobs or the cluster, but only from observing the immediate and episodic rewards while interacting with the cluster environment. We have shown that our proposed agents can outperform the baseline algorithms while optimizing both cost and time objectives, and also showcase a balanced performance while optimizing both targets. We have also discussed some key strategies discovered by the DRL agents for effective reward maximization.

Currently, we have not included the co-location goodness of different jobs which may affect the job duration further. In addition, we have not included the implications of turning the VMs on or off. However, we plan to incorporate these features with a more sophisticated reward design in the future. In addition, we want to explore how the agents behave in the actual environment with variable cluster dynamics.

Chapter 7
Conclusions and Future Directions

This chapter concludes the thesis and discusses a summary of works and key contributions. Then, it highlights several future research directions for further improvement of various resource management aspects for big data applications in Cloud environments.

7.1 Summary of Contributions

MANY prominent big data processing frameworks have emerged over the last decade due to the surge of demand for data analytics across many sectors. However, executing big data applications in the local cluster have many limitations, which can be mitigated by deploying the cluster in the Cloud. Although Cloud offers flexible, on-demand resources; shifting a big-data cluster entirely to the Cloud can be challenging as big data applications may have varying resource and performance constraints. Also, the Cloud offers a variety of VMs which varies in size and cost. Therefore, users need to be careful about managing the Cloud resources for big data applications. In a Cloud-deployed big data processing framework, these issues can be resolved by creating sophisticated resource management techniques which address various challenges. In this thesis, we have investigated resource allocation and job scheduling approaches that reduce the monetary cost of the Cloud-deployed cluster while satisfying various user and application constraints.

Chapter 1 discussed the basic concepts of big data applications and highlighted the research challenges that need to be addressed while shifting the processing of big data applications to the Cloud environment. Chapter 2 presented a taxonomy and a literature review on resource management for big data applications in Cloud environments. It

mainly focused on various resource allocation and scheduling approaches and discussed the research gaps in the existing works.

Chapter 3 presented a cost-efficient, fine-grained, and deadline-aware resource allocation mechanism. The application profiling information can be used to build the performance model of an application. This performance model can then be used to estimate an application's completion time under different parameters changes such as resource demands and input data. The chapter proposed mathematical models to build such a performance model and estimate the completion time of an application. Then this estimate was used to choose a cost-efficient resource allocation scheme.

Chapter 4 exhibited dynamic cost-efficient scheduling algorithms to schedule jobs in a Cloud-deployed big data processing cluster. It showed the optimization model which tackles all the resource capacity and demand constraints of the cluster, while the objective was to minimize cost. Then the chapter proposed two efficient job scheduling algorithms. The first algorithm utilizes the Integer Linear Programming (ILP) model to build the scheduling problem dynamically and choose the most cost-efficient scheduling decision in each scheduling iteration. The second algorithm uses a resource unification metric to find the resource capacity and demands of the VMs and jobs, respectively. Then it makes fast, cost-effective scheduling decisions by using both the resource constraints and the pricing model of the VMs. Lastly, this chapter also discussed the proposed scheduling framework for implementing new scheduling policies.

Chapter 5 presented job scheduling algorithms to be used in a cluster deployed in a hybrid Cloud, which utilizes both the pricing model and the local and Cloud VMs of a cluster effectively to reduce the monetary cost. It proposed two job scheduling algorithms. The first algorithm is based on a fast heuristic, which greedily packs executors in fewer VMs to reduce cost. The second algorithm utilizes both the pricing model and the job profile information to reduce the cost of VMs further. This chapter also showed a prototype architecture of the hybrid deployment of a big data cluster, which utilizes both local and Cloud VM effectively to reduce cost and leverage Cloud VMs in high load period of the cluster.

Chapter 6 presented efficient Deep Reinforcement Learning (DRL) based scheduling algorithms which can automatically learn various inherent cluster and job characteris-

tics to optimize one or multiple objectives. It formulated the job scheduling problem with a Reinforce Learning (RL) model. Besides, it showed how such an environment can be designed, and how the rewards can be injected into the system to accommodate multiple objectives. This chapter also used two classic DRL agents to train in the implemented prototype RL environment, to show the efficacy of the algorithms in such a cluster scheduling scenario.

7.2 Future Research Directions

In this thesis, we addressed several challenges of Cloud resource management for big data applications. However, there exist a few issues that need a more comprehensive investigation. This section gives insights into these challenges for future work in this research area.

7.2.1 Intelligent Resource Management

Machine learning algorithms are becoming more accurate and suitable for solving complex problems. Specifically, it is useful in resource management across all the different components. The resource usage statistics, system status, and the configuration parameters can be used to predict system performance. Additionally, machine learning can be used for predicting anomaly, resource demand, peak usage period, which will help to build sophisticated scheduling, resource scaling, and load-balancing algorithms. Lastly, small applications focusing on resource monitoring, performance analysis, local scheduling can be packaged as containers in a system that runs through a containerized management system to push small resource management components on the fog/Edge level for achieving faster and flexible services.

7.2.2 Application Performance Modelling in Heterogeneous Resources

The use of commodity hardware increases the availability of resources. Besides, harnessing a multi-tier deployment between Edge and Cloud can enable having emerging real-time applications. However, having a heterogeneous set of resources in a compute

cluster can be difficult to manage. Especially, the same application can behave or perform differently depending on the Edge or Cloud placement. Therefore, there is an increasing need to devise sophisticated application performance profiling methodologies and tools to investigate the effect of heterogeneous resources on each application.

7.2.3 Energy-efficient Management of Big Data Processing Systems

As big data applications use huge amount of resources, the energy consumption in the data centers is massive. To reduce the global carbon footprint, the resources in the data centers should be managed in such a way that less energy is consumed, and if possible, the use of renewable energy should be prioritized. Therefore, resource management techniques should also focus on reducing energy-consumption while managing resources for running big data applications. One approach could be the use of dynamic voltage and frequency scaling (DVFS) techniques to adjust the speed and power of servers to match with the workload demands, so that it reduces the overall energy-consumption.

7.2.4 AI-enhanced Autonomous Resource Selection and Allocation

In this thesis, we have proposed RL-based intelligent scheduling agent that can automatically learn to schedule jobs and optimize one or more target goals. However, AI-enhanced autonomic decision making has to span across different layers of resource management. In particular, AI-based approaches can be developed for resource selection and allocation, so that the resource management process can adapt to dynamic changes in the environment. Besides, we need the same AI-based solution that can be applicable to a wide range of problems, not just one.

7.2.5 Straggler Detection and Mitigation in Job Scheduling

Long jobs which may have underlying issues with performance can block a big portion of the overall resources in the cluster. In this way, it affects running the other short or critical jobs. Therefore, identifying stragglers and reducing their performance overhead

on the whole cluster is a key research area. However, stragglers can have severe performance impacts on a Cloud-deployed big data cluster as the Cloud VMs are generally heterogeneous in a sense they have diverse resource availability. In addition, having stragglers in a Cloud-deployed cluster can increase the monetary cost significantly.

7.2.6 Resource Management in Multi-tier Edge and Cloud Deployed Cluster

Many emerging applications are being developed by harnessing the potential of Edge devices. However, a huge amount of data is being generated from these devices, which is difficult to process on the Edge due to limited computing capabilities. In addition, pushing all the data to Cloud for analytics is also inefficient due to the increased latency, which might affect time-critical or real-time applications. Therefore, in a multi-tier setup spanning from Cloud to Edge, new scalable and efficient resource management techniques need to be designed to address the new challenges.

7.2.7 Improving Energy Efficiency

Fog/IoT is going to become the most investigated area in the next decade because of the availability of a vast number of wearable devices, smartphones, and smart sensors. Therefore, the distributed deployment of data processing applications will be typical. However, it is not efficient to send all the data to process in the Cloud data-centres as it might impose excessive network/transmission/bandwidth overhead in the whole system and increase the energy consumption of the data-centres. Therefore, energy-efficient software systems need to be developed that can process and analyze data on the Edge/fog level to reduce energy consumption and boost the performance of time-critical applications. Also, it will help to meet the SLA requirement through multi-tiered resource management over the Cloud data-centre, fog nodes, and mobile devices.

7.2.8 Scheduling Jobs in Geo-distributed Big Data Clusters

In this thesis, we have addressed the job scheduling problem in a cluster deployed in hybrid-Cloud. However, this problem can be further extended to a global scenario,

where the analytics jobs can be scheduled to a geo-distributed cluster. As Cloud resource pricing may vary in different regions, the monetary cost can be reduced further if the cluster is not bound to a specific region. In addition, a business spanning into multiple regions might benefit from using each region's local Cloud services. Lastly, data sensitivity and region-locks due to various company and government policies may lead to running a particular job only in a specific region.

7.2.9 Shared-sensing in Internet of Things (IoT)

As the number of IoT and mobile devices is increasing, a vast amount of resources from multiple users can be underutilized which is neither energy-efficient nor cost-effective. Therefore, IoT devices from various service providers and customers can be used collaboratively to provide efficient services. However, the software architecture should be made in such a way that it is both secure and beneficial for the collaborating partners. Besides, new protocols need to be designed on how to set the monetary cost and discount in a shared IoT infrastructure.

7.2.10 Fault Tolerant Resource Management

For time-critical applications, resource management techniques need to be fault-tolerant. If multi-tier, hybrid, heterogeneous setup are concerned, the offered flexibility and availability of resources comes with a price; increasing the chance of failures in the system. Therefore, monitoring and managing the health and condition of the cluster is important. In addition, the key metrics which indicate failure should be detected early with carefully designed Machine Learning (ML) based approaches.

7.2.11 Integrating Multiple Big Data Platforms

Depending on the diverse user demands, any application can be suitable only to a particular big data framework. Therefore, it may not be possible to port an existing application for running in a different framework. Thus, there is a need for co-existence of multiple big data processing frameworks in the same cluster. Although there are a few

cluster managers that can support multiple frameworks together within the same cluster, reource management tasks can become challenging due to the varying performance requirements of different frameworks' applications.

7.3 Final Remarks

The widespread use of big data analytics frameworks has created a massive need to deploy computing clusters in Cloud environments. In addition, Cloud service providers are also offering various types of resources and services to cater to the data analytics applications. Therefore, there is a need to create novel resource management solutions to provide monetary cost reduction while maintaining a satisfactory level of performance. In this thesis, we have proposed various resource management approaches that reduce the monetary cost of using a Cloud-deployed big data computing cluster, while satisfying various user demands such as deadline and performance constraints of the applications. In particular, we have explored resource allocation and job scheduling mechanisms that can tackle a variety of user demands and works under different cluster scenarios. The research outcomes of this thesis will also provide new opportunities to innovate robust resource management techniques for the next generation Cloud-based big data processing frameworks.

Bibliography

[1] M. D. Assunção, R. N. Calheiros, S. Bianchi, M. A. S. Netto, and R. Buyya, "Big data computing and clouds: Trends and future directions," *Journal of Parallel and Distributed Computing*, vol. 79-80, pp. 3 – 15, 2015.

[2] R. Kune, P. K. Konugurthi, A. Agarwal, R. R. Chillarige, and R. Buyya, "The anatomy of big data computing," *Software: Practice and Experience*, vol. 46, no. 1, pp. 79–105, 2016.

[3] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, p. 107113, Jan. 2008.

[4] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *IEEE International Conference on Data Mining Workshops*, 2010.

[5] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm at twitter," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 2014, pp. 170–177.

[6] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing," *Communications of the ACM*, 2016.

[7] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 38, 01 2015.

[8] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation Computer Systems*, vol. 25, pp. 599–616, 06 2009.

[9] R. Buyya, K. Ramamohanarao, C. Leckie, R. N. Calheiros, A. V. Dastjerdi, and S. Versteeg, "Big data analytics-enhanced cloud computing: Challenges, architectural elements, and future directions," in *Proceedings of the IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, 2015.

[10] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, Boston, MA, USA, 2011.

[11] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, Washington, DC, USA, 2010.

[12] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th ACM Annual Symposium on Cloud Computing*, Santa Clara, California, 2013.

[13] M. Zaharia, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, I. Stoica, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, and S. Venkataraman, "Apache spark," *Communications of the ACM*, pp. 1–5, 01 2016.

[14] L. George, *HBase: the definitive guide: random access to your planet-size data*. " O'Reilly Media, Inc.", 2011.

[15] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, p. 3540, Apr. 2010.

[16] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstrac-

tion for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, San Jose, CA, 2012.

[17] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2011.

[18] Z. Zhang, L. Cherkasova, and B. T. Loo, "Performance modeling of mapreduce jobs in heterogeneous cloud environments," in *Proceedings of the 6th IEEE International Conference on Cloud Computing (CLOUD)*, Santa Clara, CA, USA, 2013.

[19] K. Wang and M. M. H. Khan, "Performance prediction for apache spark platform," in *Proceedings of the 17th IEEE International Conference on High Performance Computing and Communications*, New York, USA, 2015.

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

[21] S. Sidhanta, W. Golab, and S. Mukhopadhyay, "Optex: A deadline-aware cost optimization model for spark," in *Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Cartagena, Colombia, 2016.

[22] A. Verma, L. Cherkasova, and R. H. Campbell, "Resource provisioning framework for mapreduce jobs with performance goals," in *Proceedings of the 12th International Middleware Conference*, 2011.

[23] O. Yildiz, S. Ibrahim, T. A. Phuong, and G. Antoniu, "Chronos: Failure-aware scheduling in shared hadoop clusters," in *2015 IEEE International Conference on Big Data (Big Data)*, 2015, pp. 313–318.

[24] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu, "Fuxi: A fault-tolerant resource management and job scheduling system at internet scale," *Proc. VLDB Endow.*, vol. 7, no. 13, p. 13931404, Aug. 2014.

[25] A. Gandhi, S. Thota, P. Dube, A. Kochut, and L. Zhang, "Autoscaling for hadoop clusters," in *2016 IEEE International Conference on Cloud Engineering (IC2E)*, 2016, pp. 109–118.

[26] L. Mashayekhy, M. M. Nejad, D. Grosu, Q. Zhang, and W. Shi, "Energy-aware scheduling of mapreduce jobs for big data applications," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 26, no. 10, pp. 2720–2733, 2015.

[27] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow : Distributed , low latency scheduling," *ACM Symposium on Operating Systems Principles (SOSP)*, p. 6984, 2013.

[28] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu, "Hopper: Decentralized speculation-aware cluster scheduling at scale," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. Sigcomm '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 379392.

[29] R. Sandhu and S. Sood, "Scheduling of big data applications on distributed cloud based on qos parameters," *Cluster Computing*, vol. 18, 12 2014.

[30] Y. Zhao, R. N. Calheiros, G. Gange, K. Ramamohanarao, and R. Buyya, "Sla-based resource scheduling for big data analytics as a service in cloud computing environments," in *2015 44th International Conference on Parallel Processing*, 2015.

[31] P. D. Kaur and I. Chana, "A resource elasticity framework for qos-aware execution of cloud applications," *Future Generation Computer Systems*, vol. 37, pp. 14–25, 2014, special Section: Innovative Methods and Algorithms for Advanced Data-Intensive Computing Special Section: Semantics, Intelligent processing and services for big data Special Section: Advances in Data-Intensive Modelling and Simulation Special Section: Hybrid Intelligence for Growing Internet and its Applications.

[32] M. Alrokayan, A. Vahid Dastjerdi, and R. Buyya, "Sla-aware provisioning and scheduling of cloud resources for big data analytics," *2014 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, 2015.

[33] N. Lim, S. Majumdar, and P. Ashwood-Smith, "A constraint programming-based resource management technique for processing mapreduce jobs with slas on clouds," *in Proceedings of the International Conference on Parallel Processing (ICPP))*, 2014.

[34] S. Maroulis, N. Zacheilas, and V. Kalogeraki, "A framework for efficient energy scheduling of spark workloads," *in Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, 2017.

[35] Q. Lu, S. Li, and W. Zhang, "Genetic algorithm based job scheduling for big data analytics," 10 2015, pp. 33–38.

[36] A. Rasooli and D. G. Down, "A hybrid scheduling approach for scalable heterogeneous hadoop systems," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, 2012, pp. 1284–1291.

[37] Y. Fonseca-Reyna, Y. Martinez, J. Cabrera, and B. Méndez-Hernández, "A reinforcement learning approach for scheduling problems," *Investigacion Operacional*, vol. 36, pp. 225–231, 01 2015.

[38] D. Nayak, V. S. Martha, D. Threm, S. Ramaswamy, S. Prince, and G. Fatimberger, "Adaptive scheduling in the cloud - sla for hadoop job scheduling," *in Proceedings of the Science and Information Conference (SAI)*, 2015.

[39] N. Zacheilas and V. Kalogeraki, "Chess: Cost-effective scheduling across multiple heterogeneous mapreduce clusters," in *in Proceedings of the IEEE International Conference on Autonomic Computing (ICAC)*, 2016.

[40] X. Zeng, S. K. Garg, Z. Wen, P. Strazdins, A. Y. Zomaya, and R. Ranjan, "Cost efficient scheduling of mapreduce applications on public clouds," *Journal of Computational Science*, vol. 26, pp. 375–388, 2018.

[41] D. Cheng, X. Zhou, P. Lama, J. Wu, and C. Jiang, "Cross-platform resource scheduling for spark and mapreduce on yarn," *IEEE Transactions on Computers*, vol. Pp, pp. 1–1, 02 2017.

[42] C. H. Chen, J. W. Lin, and S. Y. Kuo, "Deadline-constrained mapreduce scheduling based on graph modelling," in *Proceedings of the 7th IEEE International Conference on Cloud Computing (CLOUD)*, 2014.

[43] B.-G. Kim, Y. Zhang, M. Schaar, and J.-W. Lee, "Dynamic pricing and energy consumption scheduling with reinforcement learning," *IEEE Transactions on Smart Grid*, vol. 7, pp. 1–12, 11 2015.

[44] C. Imes, S. Hofmeyr, and H. Hoffmann, "Energy-efficient application resource scheduling using machine learning classifiers," in *Proceedings of the 47th International Conference on Parallel Processing*, ser. Icpp 2018. New York, NY, USA: Association for Computing Machinery, 2018.

[45] W. Zhang, S. Rajasekaran, T. Wood, and M. Zhu, "Mimp: Deadline and interference aware scheduling of hadoop virtual machines," in *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2014, pp. 394–403.

[46] S. Maroulis, N. Zacheilas, and V. Kalogeraki, "Express: Energy efficient scheduling of mixed stream and batch processing workloads," in *2017 IEEE International Conference on Autonomic Computing (ICAC)*, 2017, pp. 27–32.

[47] Z. Zong, R. Ge, and Q. Gu, "Marcher: A heterogeneous system supporting energy-aware high performance computing and big data analytics," *Big Data Research*, vol. 8, 01 2017.

[48] F. Zhang, J. Cao, W. Tan, S. Khan, K. Li, and A. Zomaya, "Evolutionary scheduling of dynamic multitasking workloads for big-data analytics in elastic cloud," *Emerging Topics in Computing, IEEE Transactions on*, vol. 2, pp. 338–351, 09 2014.

[49] E. Hwang and K. H. Kim, "Minimizing cost of virtual machines for deadline-constrained mapreduce applications in the cloud," *in Proceedings of the IEEE/ACM International Workshop on Grid Computing*, 2012.

[50] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. n. Goiri, S. Krishnan, J. Kulkarni, and S. Rao, "Morpheus: To-

wards automated slos for enterprise clusters," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2016.

[51] G. Yanfei, J. Rao, C. Jiang, and X. Zhou, "Moving mapreduce into the cloud with flexible slot management and speculative execution," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, pp. 1–1, 01 2016.

[52] A. I. Orhean, F. Pop, and I. Raicu, "New scheduling approach using reinforcement learning for heterogeneous distributed systems," *Journal of Parallel and Distributed Computing*, vol. 117, pp. 292–302, 2018.

[53] J. Polo, C. Castillo, D. Carrera, Y. Becerra, I. Whalley, M. Steinder, J. Torres, and E. Ayguadé, "Resource-aware adaptive scheduling for mapreduce clusters," in *Proceedings of the 12th ACM/IFIP/USENIX International Conference on Middleware*, 2011.

[54] K. Kc and K. Anyanwu, "Scheduling hadoop jobs to meet deadlines," in *Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science*, 2010.

[55] A. Verma, L. Cherkasova, and R. H. Campbell, "Aria: automatic resource inference and allocation for mapreduce environments," in *Proceedings of the 8th ACM International Conference on Autonomic Computing*, Karlsruhe, Germany, 2011.

[56] ——, "Resource provisioning framework for mapreduce jobs with performance goals," in *Proceedings of the 12th International Middleware Conference*, Lisbon, Portugal, 2011.

[57] A. Verma, L. Cherkasova, V. S. Kumar, and R. H. Campbell, "Deadline-based workload management for mapreduce environments: Pieces of the performance puzzle," in *IEEE Network Operations and Management Symposium*, 2012.

[58] A. Gupta, W. Xu, N. Ruiz-Juri, and K. Perrine, "A workload aware model of computational resource selection for big data applications," in *Proceedings of the IEEE International Conference on Big Data*, Washington, DC, USA, 2016.

[59] R. Tous, A. Gounaris, C. Tripiana, J. Torres, S. Girona, E. Ayguadé, J. Labarta, Y. Becerra, D. Carrera, and M. Valero, "Spark deployment and performance evaluation on the marenostrum supercomputer," in *Proceedings of the IEEE International Conference on Big Data*, Santa Clara, CA, USA, 2015.

[60] P. Petridis, A. Gounaris, and J. Torres, "Spark parameter tuning via trial-and-error," pp. 226–237, 2016.

[61] G. Wang, J. Xu, and B. He, "A novel method for tuning configuration parameters of spark based on machine learning," in *Proceedings of the 18th IEEE International Conference on High Performance Computing and Communications (HPCC)*, Sydney, Australia, 2016.

[62] A. Gounaris, G. Kougka, R. Tous, C. Tripiana, and J. Torres, "Dynamic configuration of partitioning in spark applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, pp. 1–1, 07 2017.

[63] G. P. Gibilisco, M. Li, L. Zhang, and D. Ardagna, "Stage aware performance modeling of dag based in memory analytic platforms," in *Proceedings of the 9th IEEE International Conference on Cloud Computing (CLOUD)*, San Francisco, USA, 2016.

[64] "Sparklauncher java api," https://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/launcher/package-summary.html, accessed: 2017-06-18.

[65] "The apache commons mathematics library," http://commons.apache.org/proper/commons-math/, accessed: 2017-06-18.

[66] "Joptimizer," http://www.joptimizer.com/l, accessed: 2017-06-18.

[67] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang *et al.*, "Bigdatabench: A big data benchmark suite from internet services," in *Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 488–499.

[68] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Job scheduling for multi-user mapreduce clusters," EECS Department, University of California, Berkeley, Tech. Rep. Ucb/eecs-2009-55, 2009.

[69] Q. Chen, D. Zhang, M. Guo, Q. Deng, and S. Guo, "Samr: A self-adaptive mapreduce scheduling algorithm in heterogeneous environment," in *Proceedings of the 10th IEEE International Conference on Computer and Information Technology*, 2010.

[70] C. Tian, H. Zhou, Y. He, and L. Zha, "A dynamic mapreduce scheduler for heterogeneous workloads," in *Proceedings of the 8th International Conference on Grid and Cooperative Computing*, 2009.

[71] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and qos-aware cluster management," in *Proceedings of the 19th International Conference on Architectural support for programming languages and operating systems (ASPLOS)*, 2014.

[72] S. Dimopoulos, C. Krintz, and R. Wolski, "Justice: A deadline-aware, fair-share resource allocator for implementing multi-analytics," in *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*, 2017.

[73] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *Proceedings of the USENIX Annual Technical Conference*, 2010.

[74] M. Soualhia, F. Khomh, and S. Tahar, "Task scheduling in big data platforms: A systematic literature review," *Journal of Systems and Software*, vol. 134, pp. 170–189, 09 2017.

[75] E. G. Coffman, J. Csirik, G. Galambos, S. Martello, and D. Vigo, "Bin packing approximation algorithms: Survey and classification," in *Handbook of Combinatorial Optimization*. Springer New York, 2013, pp. 455–531.

[76] J. A. Nelder and R. Mead, "A simplex method for function minimization," *The Computer Journal*, vol. 7, no. 4, pp. 308–313, 1965.

[77] G. T. Ross and R. M. Soland, "A branch and bound algorithm for the generalized assignment problem," *Mathematical Programming*, vol. 8, no. 1, pp. 91–103, 1975.

[78] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Storm at twitter," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. Acm, 2014, pp. 147–156.

[79] P. Garefalakis, K. Karanasos, P. Pietzuch, A. Suresh, and S. Rao, "Medea: Scheduling of long running applications in shared production clusters," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: Association for Computing Machinery, 2018.

[80] C. Curino, S. Krishnan, K. Karanasos, S. Rao, G. M. Fumarola, B. Huang, K. Chaliparambil, A. Suresh, Y. Chen, S. Heddaya, R. Burd, S. Sakalanaga, C. Douglas, B. Ramsey, and R. Ramakrishnan, "Hydra: a federated resource manager for datacenter scale analytics," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 177–192.

[81] D. Cheng, J. Rao, C. Jiang, and X. Zhou, "Resource and deadline-aware job scheduling in dynamic hadoop clusters," *in Proceedings of the IEEE 29th International Parallel and Distributed Processing Symposium (IPDPS)*, 2015.

[82] X. Zeng, S. Garg, Z. Wen, P. Strazdins, L. Wang, and R. Ranjan, "Sla-aware scheduling of map-reduce applications on public clouds," *in Proceedings of the 18th IEEE International Conference on High Performance Computing and Communications (HPCC)*, 2017.

[83] D. Wu, S. Sakr, L. Zhu, and H. Wu, "Towards big data analytics across multiple clusters," *in Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2017.

[84] H. Li, H. Wang, S. Fang, Y. Zou, and W. Tian, "An energy-aware scheduling algorithm for big data applications in spark," *Cluster Computing Journal*, vol. 23, 06 2020.

[85] Z. Liu, H. Zhang, and L. Wang, "Hierarchical spark: A multi-cluster big data computing framework," in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, 2017, pp. 90–97.

[86] S. Sidhanta, W. Golab, and S. Mukhopadhyay, "Deadline-aware cost optimization for spark," *IEEE Transactions on Big Data (TBD)*, 2019.

[87] H. Zhang, J. Shi, B. Deng, G. Jia, G. Han, and L. Shu, "MCTE: Minimizes task completion time and execution cost to optimize scheduling performance for smart grid cloud," *IEEE Access*, vol. 7, pp. 134 793–134 803, 2019.

[88] W.-J. Wang, Y.-S. Chang, W.-T. Lo, and Y.-K. Lee, "Adaptive scheduling for parallel tasks with QoS satisfaction for hybrid cloud environments," *The Journal of Supercomputing*, vol. 66, no. 2, pp. 783–811, Feb. 2013.

[89] V. Peláez, A. Campos, D. F. García, and J. Entrialgo, "Online scheduling of deadline-constrained bag-of-task workloads on hybrid clouds," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 19, p. e4639, May 2018.

[90] A. M. Maia, Y. Ghamri-Doudane, D. Vieira, and M. F. de Castro, "Optimized placement of scalable iot services in edge computing," in *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 2019, pp. 189–197.

[91] S. Burer and A. N. Letchford, "Non-convex mixed-integer nonlinear programming: A survey," *Surveys in Operations Research and Management Science*, vol. 17, no. 2, pp. 97–106, 2012.

[92] X. Wang, J. Wang, X. Wang, and X. Chen, "Energy and delay tradeoff for application offloading in mobile cloud computing," *IEEE Systems Journal*, vol. 11, no. 2, pp. 858–867, 2017.

[93] L. Caccetta, *Branch and Cut Methods for Mixed Integer Linear Programming Problems*. Springer US, 2000.

[94] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proceedings of the 15th ACM Workshop on Hot Top-*

ics in Networks - HotNets '16. New York, New York, USA: ACM Press, 2016, pp. 50–56.

[95] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," p. 270288, 2019.

[96] G. Rjoub, J. Bentahar, O. Abdel Wahab, and A. Bataineh, "Deep smart scheduling: A deep learning approach for automated big data scheduling over the cloud," *Proceedings - 2019 International Conference on Future Internet of Things and Cloud, FiCloud 2019*, pp. 189–196, 2019.

[97] Y. Cheng and G. Xu, "A novel task provisioning approach fusing reinforcement learning for big data," *IEEE Access*, vol. 7, pp. 143 699–143 709, 2019.

[98] L. Thamsen, J. Beilharz, V. T. Tran, S. Nedelkoski, and O. Kao, "Mary, hugo, and hugo∗: Learning to schedule distributed data-parallel processing jobs on shared clusters," *Concurrency Computation*, no. March, pp. 1–12, 2020.

[99] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," *Nature*, vol. 550, pp. 354–359, 10 2017.

[100] Y. Wei, L. Pan, S. Liu, L. Wu, and X. Meng, "Drl-scheduling: An intelligent qos-aware job scheduling framework for applications in clouds," *IEEE Access*, vol. 6, pp. 55 112–55 125, 2018.

[101] T. Li, Z. Xu, J. Tang, and Y. Wang, "Model-free control for distributed stream data processing using deep reinforcement learning," vol. 11, no. 6, p. 705718, Feb. 2018.

[102] Y. Bao, Y. Peng, and C. Wu, "Deep learning-based job placement in distributed machine learning clusters," *Proceedings - IEEE INFOCOM*, vol. 2019-April, pp. 505–513, 2019.

[103] Z. Hu, J. Tu, and B. Li, "Spear: Optimized dependency-aware task scheduling with deep reinforcement learning," *Proceedings - International Conference on Distributed Computing Systems*, vol. 2019-July, pp. 2037–2046, 2019.

[104] C. Wu, G. Xu, Y. Ding, and J. Zhao, "Explore deep neural network and reinforcement learning to large-scale tasks processing in big data," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 33, no. 13, pp. 1–29, 2019.

[105] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013.

[106] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

[107] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, no. 3-4, pp. 229–256, May 1992.