ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2018-ENST-0033

EDITE - ED 130

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

TELECOM ParisTech

Spécialité « INFORMATIQUE et RESEAUX »

présentée et soutenue publiquement par

Francesco PACE

le 18 Juin 2018

Mechanisms for Efficient and Responsive Distributed Applications in Compute Clusters

Directeur de thèse : **Pietro MICHIARDI**

Jury

Prof. Guillaume URVOY-KELLER, Université Nice Sophia Antipolis, I3S, Sophia Antipolis – France Rapporteur
Prof. Fabrice HUET, Université Côte d'Azur, CNRS, I3S, Sophia Antipolis – France Rapporteur
Dr. Marko VUKOLIC, IBM Research, Zurich – Switzerland Examinateur
Prof. Christian BONNET, EURECOM, Sophia Antipolis – France Examinateur
Prof. Melek ÖNEN, EURECOM, Sophia Antipolis – France Examinateur
Prof. Damiano CARRA, University of Verona, Verona - Italy Invite

TELECOM ParisTech
école de l'Institut Télécom - membre de ParisTech

THÈSE

Mechanisms for Efficient and Responsive Distributed Applications in Compute Clusters

Francesco PACE

Department of Data Science
Eurecom

This dissertation is submitted for the degree of
Doctor of Philosophy

Telecom Paris Tech July 2018

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Portions of the contents that appear in this dissertation have been published before in:

- Pace F., Venzano D., Carra D. and Michiardi P. [2017], Flexible scheduling of distributed analytic applications, in *Cluster, Cloud and Grid Computing (CCGRID), 2017 17th IEEE/ACM International Symposium on*, IEEE, pp. 100–109. [Pace et al., 2017]
- Pace F., Milios D., Carra D., Venzano D. and Michiardi P. [2018], Data-Driven Resource Allocation for Distributed Applications in Compute Clusters, **in submission** at *Symposium of Cloud Computing (SoCC) 2018*, ACM.
- Pace F., Milanesio M., Venzano D., Carra D. and Michiardi P. [2016], Experimental performance evaluation of cloud-based analytics-as-a-service, in *Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on*, IEEE, pp. 196–203. [Pace, Milanesio, Venzano, Carra and Michiardi, 2016]
- Vernik G., Factor M., Kolodner E. K, Michiardi P., Ofer E. and Pace F. [2018], Stocator: Providing High Performance and Fault Tolerance for Apache Spark over Object Storage, in *Cluster, Cloud and Grid Computing (CCGRID), 2018 18th IEEE/ACM International Symposium on*, IEEE. [Vernik et al., 2017]

Francesco PACE
July 2018

Acknowledgements

In the past years, many great people have been part of my life and inspired and helped my work. Without their support this thesis would not have been possible. I owe all of them a debt of gratitude and I would like to use the following lines to express my thankfulness.

First and foremost, I would like to express my gratitude to my advisor Prof. Pietro MICHIARDI for the continuous support during my Ph.D studies and related research, for his motivation and immense patient. His guidance helped me during the research and writing of this thesis. His advice on both research as well as on my career have been priceless. For me Prof. MICHIARDI was not just a mentor, he has been a good friend and helped me growing as a person; I hope to work again with him in the future. Special thanks also go to Prof. Damiano CARRA for his insightful comments and questions that helped me writing this thesis.

I would also like to thank my committee members, Prof. Guillaume URVOY-KELLER, Prof. Fabrice HUET, Dr. Marko VUKOLIC, Prof. Christian BONNET and Prof. Melek ÖNEN for serving their role even at hardship.

I also want thank the great Data Science team: thank you Raja APPUSWAMY, Rosa CANDELA, Kurt CUTAJAR, Remi DOMINGUES, Maurizio FILIPPONE, Sébastien MARMIN, Dimitrios MILIOS, Graziano MITA, Duc-Trung NGUYEN, Paolo PAPOTTI, Gia Lac TRAN and Daniele VENZANO for the amazing experience of the past years. I know that from time to time it was not easy to work with me so thank you all for being there for me when I was struggling or when I was too lazy to work. Special thanks goes to Marco MILANESIO, Daniele VENZANO and Dimitrios MILIOS for helping me developing my ideas and make sure that they see the light of the day.

Last but not the least, I would like to thank all my family: my parents and siblings for supporting me throughout this journey and in my life in general. It has not been easy!

Abstract

This dissertation addresses the problem of improving the responsiveness of distributed applications in compute clusters. We begin defining a user-defined analytic application, which is an high-level compositions of frameworks, their components and the logic necessary to carry out work. The key idea in our application definition is to distinguish classes of components, including core and elastic types: the first being required for an application to make progress, the latter contributing to reduced execution times. We show that the problem of scheduling such applications poses new challenges, which existing approaches address inefficiently. Thus, we present the design and evaluation of a novel, flexible heuristic to schedule analytic applications, that aims at high system responsiveness, by allocating resources efficiently.

However, even with an almost-optimal resource allocation, the clusters are largely under-utilized because resource allocation is based on reservation mechanisms which ignore actual resource utilization. Indeed, it is common to reserve resources for peak demand, which may occur only for a small portion of the application life time. As a consequence, cluster resources often go under-utilized. Our approach monitors resource utilization and employs a data-driven approach to resource demand forecasting, featuring quantification of uncertainty in the predictions. Using demand forecast and its confidence, our mechanism modulates cluster resources assigned to running applications, and reduces the turnaround time by more than one order of magnitude while keeping application failures under control.

To further improve the efficiency and responsiveness of distributed applications we study their performance in current cloud providers architectures. Thanks to virtualization, compute and storage clusters are more flexible, they can be easily provisioned in different sizes, and destroyed when not needed. For this reason, cloud providers architectures have a complete disaggregation between Compute and Storage which leads to the loss of data-locality, which is the notion of moving computation closer to data due to the high cost of moving data to computation. Thus, we present an intuitive notion of data locality, that we use as a proxy to rank different service compositions in terms of expected performance. Through an empirical analysis, we dissect the performance achieved by analytic workloads and unveil problems due to the impedance mismatch that arise in some configurations.

In order to solve such mismatch, we introduce Stocator, whose novel algorithm achieves both high performance and fault tolerance by taking advantage of object storage semantics. This greatly decreases the number of operations on object storage as well as enabling a much simpler approach to dealing with the eventually consistent semantics typical of object storage. Performance testing shows orders of magnitude improvements that reduce costs both for the client and the storage service provider.

Table of contents

List of figures xiii

List of tables xvii

Nomenclature xix

1 Introduction 1

1.1 Contributions . . . 5

2 Background and Related Work 7

2.1 Scheduling . . . 8

2.1.1 Computer Cluster Scheduling . . . 8

2.1.2 The problem of a reservation centric resource allocation . . . 11

2.1.3 Related Work on Scheduling for Distributed Applications . . . 11

2.2 Time-Series Analysis . . . 17

2.2.1 Autoregressive Integrated Moving Average (ARIMA) . . . 18

2.2.2 Gaussian Process Regression . . . 20

2.3 Compute and Storage Disaggregation . . . 22

2.3.1 Analytics services components . . . 22

2.3.2 Cloud Object Storage . . . 24

2.3.3 Related Work on Data Locality and Cloud Object Stores Performance . . . 25

3 A Flexible Scheduling Heuristic 29

3.1 Definitions and Problem statement . . . 30

3.1.1 Definitions . . . 30

3.1.2 Problem Statement . . . 31

3.2 A Flexible Scheduling Algorithm . . . 34

3.2.1 Design guidelines . . . 34

3.2.2 Algorithm Details . . . 35

3.2.3 Preemptive policies . . . 37

3.3 Numerical evaluation . . . 37

3.3.1 Methodology . . . 37
3.3.2 Comparison with the baseline . . . 38
3.3.3 Comparison with a malleable scheduler . . . 41
3.3.4 Comparison between different definitions of size . . . 42
3.3.5 Preemption . . . 45
3.3.6 Additional considerations . . . 45
3.4 Implementation: The Zoe system . . . 46
3.5 Experiments with Zoe . . . 48
3.6 Summary . . . 50

4 Data-Driven Resource Allocation **51**
4.1 Problem Statement . . . 52
4.2 System Design . . . 53
4.2.1 Utilization Forecasting Module . . . 54
4.2.2 Resource Shaper Module . . . 57
4.2.3 System Scalability . . . 60
4.3 Numerical Evaluation . . . 60
4.3.1 Methodology . . . 60
4.3.2 Results . . . 61
4.4 System Implementation . . . 64
4.5 Experimental Evaluation . . . 65
4.6 Summary . . . 67

5 Experimental Evaluation of Disaggregation between Compute and Storage **69**
5.1 Problem Statement . . . 70
5.2 Methodology . . . 71
5.2.1 Experimental Platform . . . 71
5.2.2 Deployment scenarios . . . 72
5.2.3 Compute-to-Data path . . . 73
5.2.4 Benchmark and Workloads . . . 75
5.2.5 Performance metrics . . . 75
5.3 Results . . . 76
5.3.1 Analytics Application Benchmark . . . 76
5.3.2 Summary of the results . . . 81
5.4 Summary . . . 82

6 Stocator: High Performance Connector for Object Stores **83**
6.1 Apache Spark . . . 84
6.2 Motivation . . . 86
6.3 Stocator Logic . . . 87

6.3.1 Basic Stocator protocol . . . 88
6.3.2 Alternatives for reading an input dataset . . . 88
6.3.3 Streaming of output . . . 89
6.3.4 Optimizing the read path . . . 89
6.3.5 Examples . . . 91
6.4 Methodology . . . 92
6.4.1 Experimental Platform . . . 92
6.4.2 Deployment scenarios . . . 92
6.4.3 Benchmark and Workloads . . . 93
6.4.4 Performance metrics . . . 94
6.5 Experimental Evaluation . . . 94
6.5.1 Reduction in run time . . . 94
6.5.2 Reduction in the number of REST calls . . . 98
6.6 Summary . . . 100

7 Conclusions and Perspectives 101

References 103

Appendix A French Résumé 111
A.1 Introduction . . . 111
A.2 Une heuristique de planification flexible . . . 114
A.2.1 Définitions et énoncé du problème . . . 116
A.2.2 Un algorithme de planification flexible . . . 120
A.2.3 Résumé . . . 121
A.3 Allocation de ressources basée sur les données . . . 121
A.3.1 Énoncé du problème . . . 122
A.3.2 Conception du système . . . 123
A.3.3 Résumé . . . 124
A.4 Évaluation expérimentale de la désagrégation entre calcul et Espace de rangement . . . 125
A.4.1 Énoncé du problème . . . 126
A.4.2 Scénarios de déploiement . . . 126
A.4.3 chemin Compute-to-Data . . . 127
A.4.4 Résumé . . . 128
A.5 Stocator: connecteur haute performance pour les magasins d'objets for Object Stores . . . 130
A.5.1 Motivation . . . 131
A.5.2 Résumé . . . 133
A.6 Conclusions et Perspectives . . . 133

List of figures

1.1 Boxplot showing Resource Utilization from Google Cluster [Reiss et al., 2012; Wilkes, 2011]. 2

2.1 Time-Series Example. 18
2.2 Logical components in an analytics cluster. 23

3.1 Illustrative examples of request scheduling: (top) rigid, (middle) malleable, (bottom) flexible approaches. 33
3.2 Workload Definition: CDFs of different metrics. 39
3.3 Comparison of turnaround and queue time distributions, and application slowdown distributions for FIFO and SJF policies. White boxes (right box of every pair) corresponds to our flexible scheduler, gray boxes correspond to the baseline. *B-E* stands for batch elastic and *B-R* stands for batch rigid applications. 40
3.4 Comparison of queues size for FIFO and SJF between our flexible scheduler and the baseline. The white boxes (right box of every group) correspond to our flexible algorithm, gray boxes to the baseline. 40
3.5 Comparison of resource allocation distributions for FIFO and SJF policies, between our flexible scheduler and the baseline. White boxes (right box of every pair) correspond to our approach, dashed boxes to the baseline. 41
3.6 Comparison of turnaround and queue time distributions, and application slowdown distributions for the SJF policy with different definitions of size. White boxes (right box of every pair) corresponds to our flexible scheduler, gray boxes correspond to the baseline. *B-E* stands for batch elastic and *B-R* stands for batch rigid applications. . . . 42
3.7 Comparison of queues size for the SJF policy with different definitions of size. White boxes (right box of every pair) corresponds to our flexible scheduler, gray boxes correspond to the baseline. 43
3.8 Comparison of resource allocation distributions for the SJF policy with different definitions of size. White boxes (right box of every pair) corresponds to our flexible scheduler, gray boxes correspond to the baseline. 44

3.9 On the left, comparison of queuing time distributions between scheduling with and without preemption. White boxes (left box of every pair) correspond to a *non-preemptive* system, gray boxes to our preemptive algorithm. On the right, turnaround ratio distributions between scheduling with and without preemption. *B-E* stands for batch elastic applications, *B-R* stands for batch rigid applications and *Int* is for interactive applications. 45
3.10 Comparison of turnaround time distributions using the FIFO discipline. White boxes (right box of every pair) correspond to the second generation of Zoe that implements our algorithm. *B-E* stands for batch elastic and *B-R* stands for batch rigid applications. 49

4.1 System overview: shaded boxes represent existing components, white boxes indicate new components presented in this work. 53
4.2 Boxplot showing error distribution of predicted utilization for a collection of time series in our academic cluster with different history points and, in case of Gaussian Process (GP), different kernels. The red triangle is the mean. 56
4.3 Boxplots comparing baseline vs optimistic vs pessimistic approaches over different metrics, using an oracle in place of the prediction module. The red triangle is the mean. 62
4.4 Heat maps showing the effect of K_1 and K_2, which compose β, on different metrics when using Autoregressive Integrated Moving Average (ARIMA) and GP. Bright cells are better. 63
4.5 Boxplots comparing baseline vs pessimistic dynamic approach over memory slack and turnaround time distributions using GP-based resource shaping. The red triangle is the mean. 66

5.1 *Compute-to-Data path* for different scenarios during read operations. 73
5.2 *Compute-to-Data path* for different scenarios during write operations. 74
5.3 Resource utilization with the TPC-DS workload in different scenarios. The ticks on the X axis “W", “D" and “S" stand, respectively for worker, datanode and Swift machines. The resource utilization reported is a global average across all instances of each layer. The network utilization between the GC-V and GC scenarios is similar because volumes' network is opaque to the Operating System and, therefore, counted in the disk utilization. 79
5.4 DFSIO and TPC-DS CDFs for task runtimes in all scenarios. 80

6.1 Hadoop Storage Connectors 85
6.2 A Spark program that executes a single task that produces a single output object. . . . 87
6.3 A Spark program where three tasks each write an object part. 91
6.4 Benchmarks REST calls comparison 95
6.5 Object Storage bytes read/written comparison 99

A.1 Exemples d'ordonnancement des demandes: (haut) rigide, (moyen) malléable, (bas) flexible approches. 119

A.2 Vue d'ensemble du système: les boîtes ombrées représentent les composants existants, les boîtes blanches indiquent les nouvelles composants présentés dans ce travail. 123
A.3 Programme Spark qui exécute une seule tâche produisant un seul objet. 132

List of tables

3.1 Comparison of average turnaround, CPU and Memory allocation for FIFO and SJF policies, between our flexible and a malleable scheduler. 42
3.2 Definition of size used in the evaluation 43

5.1 Expected scenarios' performance ranking. 75
5.2 Workloads' details. 76
5.3 Analytics applications benchmark results in ascending order. 77

6.1 Breakdown of REST operations by type for the Spark program that creates an output consisting of a single object. 87
6.2 Possible operations performed by the Spark application showed in fig. 6.3 90
6.3 Workload speedups when using Stocator 96
6.4 Ratio of REST calls compared to Stocator 96
6.5 Financial cost for REST calls compared to Stocator for IBM, AWS, Google and Azure infrastructure 97

A.1 Ventilation des opérations REST par type pour le programme Spark qui crée un seul objet. 132

Nomenclature

Acronyms / Abbreviations

AaaS Analytics-as-a-Service

ARIMA Autoregressive Integrated Moving Average

AWS Amazon Web Services

ETL Extract, Transform and Load

FCS Feedback Control Scheduling

FIFO First-In-First-Out

GC Garbage Collector

GP Gaussian Process

HDFS Hadoop Distributed File System

HMRCC Hadoop Map Reduce Client Core

HPC High-Performance Computing

JVM Java Virtual Machine

ML Machine Learning

MSE Mean Squared Error

OOM Out Of Memory

OSD Object Storage Daemon

OS Operating System

QOS Quality of Service

RPC Remote Procedure Call

SJF Shortest Job First

SLA Service Level Agreement

SLO Service Level Objective

VM Virtual Machine

Chapter 1

Introduction

The last decade has witnessed the proliferation of numerous distributed frameworks to address a variety of large-scale data analytics and processing tasks. First, MapReduce [Dean and Ghemawat, 2008] has been introduced to facilitate the processing of bulk data. Subsequently, more flexible tools, such as Dryad [Isard et al., 2007], Spark [Zaharia et al., 2012], Flink[1] and Naiad [Murray et al., 2013], to name a few, have been conceived to address the limitations and rigidity of the MapReduce programming model. Similarly, specialized libraries such as MLLib [Meng et al., 2016] and systems like TensorFlow [Abadi et al., 2016] have seen the light to cope with large-scale machine learning problems. In addition to a fast growing ecosystem, individual frameworks are driven by a fast-pace development model, with new releases every few months, introducing substantial performance improvements. Since each framework addresses specific needs, a wide choice of tools and combinations thereof are available to users that now can address the various stages of their data analytics projects.

This context has driven a lot of research [Delimitrou and Kozyrakis, 2013, 2014; Delimitrou et al., 2015; Ghit and Epema, 2016; Hindman et al., 2011; Isard et al., 2009; Kuzmanovska et al., 2016; Ousterhout et al., 2013; Schwarzkopf et al., 2013; Vavilapalli et al., 2013; Verma et al., 2015] in the area of resource allocation and scheduling, from both the academia and the industry. These efforts materialize in cluster management systems that offer simple mechanisms for users to request the deployment of the framework they need. The general underlying idea is that of sharing cluster resources among a heterogeneous set of frameworks, as opposed to static partitioning, which has been dismissed for it entails low resource allocation [Hindman et al., 2011; Schwarzkopf et al., 2013; Verma et al., 2015]. Existing systems divide the resources at different levels. Some of them, e.g. Mesos and YARN, target *low-level* orchestration of distributed computing frameworks: to this aim, they require non-trivial modifications of such frameworks to operate correctly. Others, e.g. Kubernetes[2] and Docker Swarm[3], focus on provisioning and deployment of containers, and are thus oblivious to the characteristics of the frameworks running in such containers.

[1] https://flink.apache.org/
[2] http://kubernetes.io/
[3] https://docs.docker.com/swarm/

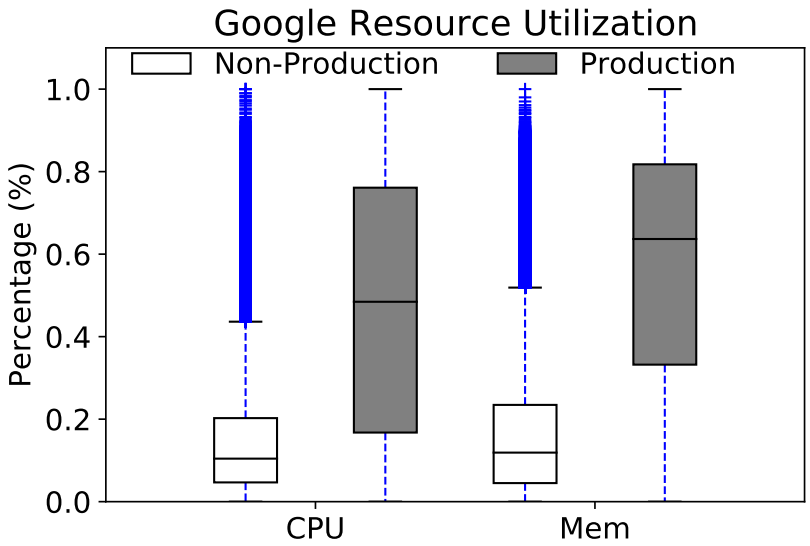


Fig. 1.1 Boxplot showing Resource Utilization from Google Cluster [Reiss et al., 2012; Wilkes, 2011].

Despite such efforts, data-center resources go often under utilized, as shown in recent traces from large-scale production deployments [Reiss et al., 2012; Wilkes, 2011]: Figure 1.1 illustrates resource utilization in a operational cluster at Google, for a mixed workload of production services and batch applications; in most cases ($\sim$ 80%) resource utilization is less than 40% or 80% of the allocated resources depending on different application types[4].

Current approaches that address efficiency requirements fall in two broad categories. The first category involves methodologies that aim at steering tenants' behavior through the design of incentive mechanisms; tenants are endowed with the task of optimizing their cost to operate their applications, whereas providers operate on prices to steer the allocation of idle resources. Such approaches are largely adopted by public cloud providers [Babaioff et al., 2017]. The second category concerns approaches that operate at the system level, and propose mechanisms that allocate resources based on tenants' reservations[5][6] [Ghodsi et al., 2011; Hindman et al., 2011; Rasley et al., 2016; Schwarzkopf et al., 2013; Verma et al., 2015].

The ultimate goal of the above line of research is to render the concept of resource reservation obsolete, and either let tenants reason in terms of value and cost [Babaioff et al., 2017], or let the system determine how to avoid wasting precious and costly resources, especially when the latter are scarce and entail application queuing in the scheduler.

A major scheduling constraint that research in the area of resource allocation and scheduling has to face is about data-locality, which refers to the ability to move the computation close to where the actual

[4] In the analysis we saw that some applications were using more resources than requested and this was confirmed by Google staff. Their system allows the user to go above the reservation when resources are available. Since not all the systems can do this (e.g.; Docker), we decided to remove that portion of the data.

[5] http://www.docker.com/

[6] https://aws.amazon.com/emr/

data resides, instead of moving large data to computation. This minimizes network congestion and increases the overall throughput of the system. Whereas before it was possible to put the task anywhere, now it needs to go on one of the data replicas.

However nowadays thanks to virtualization, compute and storage clusters are more flexible, they can be easily provisioned in different sizes, and destroyed when not needed[7]. Increasingly, such storage and processing systems are exposed to users as *services*, deployed on either public or private cloud computing environments, rather than on bare-metal machines in private clusters. Indeed, many companies offer Analytics-as-a-Service (AaaS) clusters to run a variety of applications: Amazon Web Services (AWS) with Elastic MapReduce[8], DataBricks Cloud[9], Cloudera Cloud[10] and Google Cloud Hadoop[11] are noteworthy examples.

In cloud computing environments, the architecture of analytics clusters is the result of the composition of several services, consisting of three (logically separated) layers: the *Compute layer* refers to all cluster nodes that run the data processing application (e.g., a Spark application); the *Data layer* refers to any combination of storage services (e.g., HDFS[12] or Swift[13]); and the *Storage layer* that physically stores the data, including ephemeral disks, object and elastic block stores.

In addition, it is likely for the Data or Storage layers and the Compute layer to be on different racks or even data-centers: as a consequence, the traditional wisdom of *data locality* may be challenged. For example, consider Amazon S3[14]: data resides on a set of machines dedicated just to storage, breaking data locality completely.

Currently, users of AaaS have abundant information about pricing and about the durability of resources. It is possible to reason about cost-based service dimensioning, and to select appropriate storage services depending on data availability and durability objectives. As a consequence, it is today possible to build data ingestion, storage and processing pipelines, by composing – in various combinations – the three layers defined above.

The questions that we address in this thesis is: what happens to the performance, and to the completion time in particular, of analytics applications with different type of Compute and Data layer configurations?

During our quest to answer this question, we discover the existence of some impedance mismatch between large-scale frameworks and a data storage solution, called cloud object storage, that is currently widely used among providers; Amazon S3, Azure Blob storage[15], and IBM Cloud Object Storage[16], are highly scalable distributed cloud storage systems offering high capacity and cost effective storage. Large-

[7] https://aws.amazon.com/application-hosting/benefits/
[8] https://aws.amazon.com/emr/
[9] Solution hosted on AWS: https://databricks.com/product/databricks-cloud
[10] Solution hosted on AWS: http://www.cloudera.com/content/cloudera/en/solutions/partner/Amazon-Web-Services.html
[11] https://cloud.google.com/hadoop/
[12] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[13] http://docs.openstack.org/developer/swift/development_saio.html
[14] https://aws.amazon.com/s3/
[15] https://azure.microsoft.com/en-us/pricing/details/storage/blobs/
[16] https://www.ibm.com/cloud-computing/products/storage/object-storage/cloud/

scale computing frameworks were originally designed to work on data stored in Hadoop Distributed File System (HDFS)[17] where the storage and processing are co-located in the same server cluster. Moving data from object storage to HDFS in order to process it and then moving the results back to object storage for long term storage is inefficient.

Until now Hadoop connectors to object storage, e.g., S3a[18] and the Hadoop Swift Connector[19], have been based on file semantics, a natural assumption given that their model of operation is based on the way that Hadoop interacts with its original storage system, HDFS. However, treating object storage like a file system constitutes an impedance mismatch, which can lead to poor performance and incorrect execution. In particular, operations that are atomic for files may not be atomic for objects and operations that are inexpensive for files may not be inexpensive for objects, and vice versa. For example, to rename a directory in a file system requires a single atomic operation, whereas in object storage it requires copy and delete operations for each of the objects in the tree under the "virtual directory"[20].

We are not the first to recognize the poor performance of the object storage connectors. Others have tried to improve performance, by sacrificing speculative execution, and then writing objects directly to their final names, e.g., the DirectOutputCommitter[21] for Amazon S3, or by renaming Hadoop output objects to their final names when tasks complete (task commit) instead of waiting until the entire job completes (job commit)[22]. However, due to the impedance mismatch these attempts led to subtle failures.

Current connectors can also lead to failures and incorrect execution because the list operation on object storage containers/buckets is eventually consistent. EMRFS[23] from Amazon and S3mper[24] from Netflix overcome eventual consistency by storing file metadata in DynamoDB[25], an additional strongly consistent storage system separate from the object store. A similar feature called S3Guard[26] is being developed by the Hadoop open source community for the S3a connector. Solutions like these, which require multiple storage systems, are complex and can introduce issues of consistency between the stores. They also add cost since users must pay for the additional strongly consistent storage.

The goal of this dissertation is to improve systems responsiveness in private and public cloud providers, applying novel scheduling techniques and leveraging existing machine learning approaches to steer the behavior of scheduling decisions. Using such approach improves resource utilization by $\sim$50% and average turnaround time, which is the time that an application reside inside the system, by more than two orders of magnitude.

[17] https://hortonworks.com/apache/hdfs/
[18] https://aws.amazon.com/sdk-for-java/
[19] https://github.com/openstack/sahara-extra/tree/master/hadoop-swiftfs
[20] Object stores emulate directories through hierarchical naming.
[21] https://github.com/apache/spark/pull/12229
[22] https://issues.apache.org/jira/browse/MAPREDUCE-6336
[23] https://aws.amazon.com/blogs/aws/emr-consistent-file-system/
[24] http://techblog.netflix.com/2014/01/s3mper-consistency-in-cloud.html
[25] https://aws.amazon.com/dynamodb/
[26] http://www.slideshare.net/hortonworks/s3guard-whats-in-your-consistency-model

1.1 Contributions

We start by giving, in **Chapter 2** some background and related work required to understand the technical details of this thesis. We briefly introduce the notion of scheduling and how it evolved in the last 70 years, then we focus our attention toward compute clusters, the different types of existing cluster scheduler architectures, related work for scheduling and resource allocation of distributed applications and we highlight a common problem, which revolves around the way that users are able to express their resources constraints. To address this issue, we leverage methodologies developed in the field of time-series analysis and Machine Learning (ML). We define a time-series and its characteristics in the context of this thesis. Then we focus on time-series forecasting and its methodology. In particular we introduce two techniques that are widely adopted to forecast future values. Finally, we give an overview of the disaggregation between compute and storage, highlighting the different analytic components and focusing one storage technology that is currently wide spread: cloud object for storage.

The endeavor of **Chapter 3** is to fill the gap that exists in current approaches, and *raise the level of abstraction* at which scheduling works. We introduce a general and flexible definition of *applications*, how they are composed, and how to execute them. For example, a user application addressing the training of a statistical model involves: a user-defined program implementing a learning algorithm, a framework (e.g., Spark) to execute such a program together with information about its resource requirements, the location for input and output data and possibly parameters exposed as application arguments. Users should be able to express, in a simple way, how such an application must be packaged and executed, submit it, and expect results as soon as possible. We show that scheduling such applications represents a departure from what has been studied in the scheduling literature, and we present the design of a new algorithm to address the problem. A key insight of our approach is to exploit the properties of the frameworks used by an application, and distinguish their components according to classes, core and elastic: the first being required for an application to produce work, the latter contributing to reduced execution times.

In **Chapter 4** we present a system that dynamically allocates resources according to historical observations of its utilization. More specifically, we leverage a machine learning algorithm in order to adjust allocation to the expected utilization. We present our design of a data-driven scheduling mechanism that improves cluster utilization, thus decreasing the average turnaround time, while preventing application failures due to resource contention. Our approach monitors resource utilization and relies on sophisticated online resource demand forecasting to modulate allocated resources such as they approximate utilization patterns well. Our experiments, that we conduct on a system simulator as well as a prototype implementation using real-life data-center traces, indicate substantial gains over existing alternatives: our approach contributes to more efficient and responsive clusters, while carefully controlling the number of application failures due to the approximate nature of our control approach.

In **Chapter 5** we take an experimental approach, and propose a measurement methodology and campaign, whose objective is to analyze the performance corresponding to an intuitive notion of distance between where computation happens and data reside. In doing so, we define an extensive set of

application workloads that challenge the systems under study in different ways. Ultimately, our goal is to overcome the limitations of prior works that only provide a boolean vision of data locality: our results indicate that – in general – the intuitive distance metric we present in this thesis is a good proxy to reason about performance ranking. However, impedance mismatch between different services and application workloads must be taken into account to formulate plausible explanations for outliers in terms of performance.

In **Chapter 6**, we introduce Stocator, whose novel algorithms achieve both high performance and fault tolerance by taking advantage of object storage semantics. This greatly decreases the number of operations on object storage as well as enabling a much simpler approach to dealing with the eventually consistent semantics typical of object storage. We have implemented our connector for both the OpenStack Swift API[27] and the Amazon S3 API, and have shared it in open source[28]. We have compared its performance with the S3a and Hadoop Swift connectors over a range of workloads and found that it executes far less operations on the object store, in some cases as little as one thirtieth of the operations. Since the price for an object storage service typically includes charges based on the number of operations executed, this reduction in the number of operations lowers costs in addition to reducing the load on client software. It also reduces costs and load for the object storage provider since it can serve more clients with the same amount of processing power. Stocator also substantially increases performance for Spark workloads running over object storage, especially for write intensive workloads, where it is as much as 18 times faster. Stocator is in production in the IBM Cloud and has enabled the SETI project to perform computationally intensive Spark workloads on multi-terabyte binary signal files[29].

A summary of technical contributions for this dissertation can be found in **Chapter 7**.

[27] https://developer.openstack.org/api-ref/object-storage/
[28] https://github.com/SparkTC/stocator
[29] https://medium.com/ibm-watson-data-lab/simulating-e-t-e34f4fa7a4f0

Chapter 2

Background and Related Work

In this Chapter we give the necessary background information for Chapters 3 to 6. It is divided in three sections: *scheduling*, *time-series analysis* and *compute-storage disaggregation*.

Our objective is *to improve the efficiency and performance of distributed applications in compute cluster*. This is partially achieved by improving the allocation of scarce resources to increase both the responsiveness and utilization of the system which is imperative for both the providers and their users. In section 2.1 we briefly introduce what scheduling is and how it evolved in the last 70-year, then we focus our attention toward compute clusters, the different types of existing cluster scheduler architectures, related work and we highlight (in section 2.1.2) a common issue, which revolves around the way that users are able to express their resources constraints.

To tackle such issue, we leverage methodologies developed in the field of time-series analysis and Machine Learning (ML). In section 2.2 we provide the definition of time-series and its characteristics in the context of this thesis. Then we focus on time-series forecasting and its methodology. In particular we briefly introduce two techniques that are widely adopted to forecast future values: *(i)* Autoregressive Integrated Moving Average (ARIMA) (section 2.2.1) and *(ii)* Gaussian Process (GP) regression (section 2.2.2).

To further improve the efficiency and responsiveness of distributed applications we study their performance in current cloud providers architectures. Thanks to virtualization, compute and storage clusters are more flexible, they can be easily provisioned in different sizes, and destroyed when not needed. We are seeing a trend in which *(i)* providers tend to disaggregate storage and compute cluster and *(ii)* new technologies (e.g., IBM Storlets [Rabinovici-Cohen et al., 2014]) are being developed to ship part of the code toward the storage, reducing the data transmitted to the compute [Moatti et al., 2017]. This disaggregation leads to the loss of data-locality, which is the notion of moving computation closer to data due to the high cost of moving data to computation. In section 2.3 we give an overview of the disaggregation between compute and storage, highlighting the different analytic components and focusing one storage technology that is currently wide spread: cloud object for storage (section 2.3.2). Finally we provide some related work for data locality and performances of cloud object stores.

2.1 Scheduling

Scheduling is concerned with allocation of resources to activities with the objective of optimizing one or more performance measures. Depending on the situation, resources and activities can take on many different forms. Resources may be cashiers at a supermarket, machines in an assembly plant, CPU, memory and I/O devices in a computer system, runways at an airport, mechanics in an automobile repair shop, etc. Activities may be various customers, operations in a manufacturing process, execution of a computer program, landings and take-offs at an airport, car repairs in an automobile repair shop, and so on. There are also many different performance measures to optimize through a scheduling algorithm. One objective may be the minimization of the makespan, also known as turnaround, which is the total time that elapses from the beginning to the end of the activity, while another objective may be the minimization of the number of late jobs.

The study of scheduling starts in 1950s; researchers in industrial engineering, operations research, and management faced the problem of managing various activities occurring in a workshop. Good scheduling algorithms are able to lower the production cost in a manufacturing process, enabling companies to stay competitive. At the end of 1960s, computer scientists encountered scheduling problems in the development of operating systems; in those days, computational resources (such as CPU, memory and I/O devices) were scarce. Efficient utilization of such scarce resources can reduce the cost of executing computer programs leading to a economic reason for the study of scheduling.

In the 1950s, the scheduling problems were relatively simple; a number of efficient algorithms have been developed to provide optimal solutions. However, as time went by, the problems became more sophisticated, and researchers were unable to develop efficient algorithms for them. With the advent of complexity theory, researchers started to realize that many of these problems may be inherently difficult to solve. In fact, by the 1970s, researchers found that many scheduling problems are NP-hard in their complexity. In the 1980s, several different directions were pursued both in academia and industry; one was the development and analysis of approximation algorithms while another was the increasing focus on stochastic scheduling problems. After the 1980s, research in scheduling theory took off quickly.

2.1.1 Computer Cluster Scheduling

A **computer cluster** is a set of loosely or tightly connected computers that work together so that they can be viewed as a single system. Clusters are usually deployed to improve performance and availability over that of a single computer and emerged as a result of convergence of a number of computing trends including the availability of low-cost microprocessors, high-speed networks, and software for high-performance distributed computing.

Large-scale clusters are expensive, thus scheduling is an important topic to improve utilization and efficiency, which are the key indicators for good resource management. Cluster schedulers have evolved significantly in the last few years; their architecture has moved from monolithic designs to much more flexible, disaggregated and distributed designs. Next we highlight the most common architectures used to schedule work inside such clusters.

Monolithic schedulers Many cluster schedulers - such as most High-Performance Computing (HPC) schedulers, various early Hadoop [Shvachko et al., 2010] schedulers, Borg [Verma et al., 2015] and the Kubernetes [Burns et al., 2016] scheduler – are monolithic. A single scheduler process runs on one machine and assigns tasks to machines. The entire workload is handled by the same scheduler, and all tasks run through the same scheduling logic. This simple approach led to increasingly sophisticated schedulers being developed. For example, the Paragon [Delimitrou and Kozyrakis, 2013] and Quasar [Delimitrou and Kozyrakis, 2014] schedulers, which use a machine learning approach to avoid negative interference between jobs competing for resources.

However nowadays, most clusters run different types of applications (as opposed to just Hadoop MapReduce [Dean and Ghemawat, 2008] jobs in the early days). Thus, maintaining a single scheduler implementation that handles heterogeneous workloads can be tricky, for several reasons:

1. A scheduler might want to treat long-running service jobs and batch analytics jobs differently.
2. Different applications have different needs; thus supporting them all is not an easy tasks. By adding features to the scheduler, we also increasing the complexity of its logic and implementation.
3. The order in which the scheduler processes tasks becomes an issue: queuing effects (e.g., head-of-line blocking) and backlog can become an issue unless the scheduler is carefully designed.

Two-Leveling scheduling Two-level scheduling architectures address some of the previous problems by separating the concerns of **resource allocation**, that is *the assignment of available resources*, and **task placement**, that is *where to place the task based on the requirements specified in the task definition*. This allows the task placement logic to be tailored towards specific applications, but also maintains the ability to share the cluster between them. The Mesos [Hindman et al., 2011] cluster manager pioneered this approach, and YARN [Vavilapalli et al., 2013] supports a limited version of it. In Mesos, resources are directly offered to application-level schedulers, while YARN allows the application-level schedulers to request resources instead. The general idea is: workload-specific schedulers interact with a resource manager that creates dynamic partitions of the cluster resources for each workload. This is a very flexible approach that allows custom, workload-specific scheduling policies.

Yet, the separation of concerns in two-level architectures has a drawback: the application-level schedulers lose omniscience, i.e., they cannot see all the possible placement options any more. Instead, they merely see those options that correspond to resources offered (Mesos) or allocated (YARN) by the resource manager. This has one big disadvantage: application-specific schedulers care about many different aspects of the underlying resources, but their only means of choosing resources is the offer/request interface with the resource manager; this interface can easily become quite complex.

Shared-State scheduling Shared-state architectures move to a semi-distributed model, in which several replicas of cluster state are independently updated by application-level schedulers. After the change is applied locally, the scheduler issues an optimistically concurrent transaction to update the

shared cluster state. In some cases, this transaction may fail: another scheduler may have made a conflicting change in the meantime. The base principle of these schedulers is that these fails occur rarely, and when they do the scheduling decision is rolled-back and repeated.

The most prominent examples of this type of scheduling are Omega [Schwarzkopf et al., 2013], and Apollo [Boutin et al., 2014]. All of these materialize the shared cluster state in a single location: the *cell state* in Omega and the *resource monitor* in Apollo. Apollo differs from Omega as its shared-state is read-only, and the scheduling transactions are submitted directly to the cluster machines. The machines themselves check for conflicts and accept or reject the changes. This allows Apollo to make progress even if the shared-state is temporarily unavailable.

Shared-state architectures have some drawbacks too: unlike a centralized scheduler, they must work with stale information, and they may experience degraded scheduler performance under high contention [Schwarzkopf et al., 2013].

Fully-Distributed scheduling Fully-distributed architectures take the disaggregation even further: there is no coordination between schedulers, and they use many independent schedulers to service the incoming workload. Each of these schedulers works only with its local view of the cluster which in most cases is partial and out-of-date. Jobs can typically be submitted to any scheduler, and each scheduler may place tasks anywhere in the cluster. Unlike with two-level schedulers, there are no partitions that each scheduler is responsible for.

Sparrow [Ousterhout et al., 2013] was one of the first distributed scheduler, although the underlying concept first appeared in 1996. The key foundation of Sparrow is based on an hypothesis that the tasks we run on clusters are becoming ever shorter in duration, supported by an argument that fine-grained tasks have many benefits. Consequently, the authors assume that tasks are becoming more numerous, meaning that a higher decision throughput must be supported by the scheduler. Since a single scheduler may not be able to keep up with this throughput (assumed to be a million tasks per second), Sparrow spreads the load across many schedulers.

Some of the drawbacks include: *(i)* they cannot support or afford complex or application-specific scheduling policies because of their rapid decision design which is based on minimal information and *(ii)* have difficulty enforcing global invariants since there is no central control.

Hybrid scheduling Hybrid architectures are a recent design that seeks to address the drawbacks of fully distributed architectures by combining them with monolithic or shared-state designs. For example, in Tarcil [Delimitrou et al., 2015] and Hawk [Delgado et al., 2015], there are two scheduling paths: a distributed one for part of the workload (e.g., very short tasks, or low-priority batch workloads), and a centralized one for the rest. The behavior of each constituent part of a hybrid scheduler is identical to the part's architecture described above. However, to the best of our knowledge, no hybrid scheduler have been deployed in production.

2.1.2 The problem of a reservation centric resource allocation

Users gain access to computing resources by specifying the amount required to run their application, in the form of a reservation request. Upon receiving a request, the cluster **scheduler** decides which application to serve based on the scheduling policy the provider implements (e.g.; First-In-First-Out (FIFO), Shortest Job First (SJF)). Cluster schedulers operate according to several variants of objective functions, the most common being the *(i)* **average turnaround time** (also called make span or completion time) and *(ii)* **cluster utilization** [Leung, 2004; Schwarzkopf et al., 2013]. The first metric accounts for the average time requests spend in the system (queuing and execution times). The second metric considers the utilization of the available resources. Optimizing for such objectives translates into high system responsiveness, which is desirable from both tenant and provider perspective.

Cluster schedulers use a resource management mechanism that is in charge of resource provisioning and management. Given a **resource request**, the resource manager determines its admission in the cluster based on its **reservation** information.[1] An admitted request triggers a **resource allocation** procedure, which eventually [Schwarzkopf et al., 2013] concludes with reserved resources being exclusively allocated to the request.

In most system implementations, the concept of reservation and allocation coincide, although neither is representative of the true **resource utilization** a request might induce on the system. In fact, resource utilization is generally not constant throughout a request lifetime, and fluctuates according to application behavior [Yan et al., 2016].

The main consequence for current cloud environments is that reservation requests are **engineered to cope with peak resource demands** of an application. This is a key factor that induces poor system utilization, and ultimately, efficiency. This condition is exacerbated by coarse-grained reservation specifications, which is a common practice in public cloud providers: instance *flavors* exhibit discrete gaps in terms of resource units. In fact, picking the right configuration for cloud applications (and in particular for the "big data" applications) is a daunting task [Alipourfard et al., 2017], which requires sophisticated optimization mechanisms going beyond human tuning abilities.

Thus, mechanisms to reduce **resource slack**, which is defined as the difference between resource allocation and utilization, are truly needed, for they can prevent clusters from denying admission for new requests which would queue up, while spare capacity goes unused.

2.1.3 Related Work on Scheduling for Distributed Applications

While we cannot do justice to the richness of the scheduling literature, in this section we organize related work in two groups. The first group covers works that studied different scheduling approaches with a heterogeneous set of applications without dealing with the problem of reservation centric resource allocation; while the second group studied different ways to deal with it and focus on ways to dynamically adjust resource allocation.

[1] In our prose, we neglect several important technical details that are however irrelevant to our point, such as quota management, security aspects, and concurrency control, to name a few.

Scheduling Heterogeneous Set of Applications

Many systems have been designed to cope with the problem of sharing cluster resources across a heterogeneous set of applications, some of which can be tweaked to achieve the goals we set in this thesis. For example, Yarn [Vavilapalli et al., 2013] and Mesos [Hindman et al., 2011] have been among the first to enable multiple frameworks to coexist in the same cluster: usage of these "two-level" schedulers yields a big improvement as compared with monolithic approaches to resource scheduling. Originally designed for analytic frameworks, such systems deal with the scheduling of low-level processing tasks.

Apache Mesos is a two-level scheduler that splits the resource management and placement functions between a central resource manager (that makes resource offers to application frameworks) and multiple frameworks such as Hadoop and Spark using an offer-based mechanism. Each framework has then an individual scheduler that handles its assigned resources.

Yarn, in contrast with Mesos, takes a request-based approach and considers application specific constraints to allocate resources from a fixed set of machines to each application.

Recently, more general approaches address the problem of cluster-wide resource management: Omega [Schwarzkopf et al., 2013] reason at the "container" level, and are optimized to achieve efficient placement and allocation of cluster resources, when absorbing a very heterogeneous workload. This latter includes a majority of *long-running services*, which power Web-scale, latency-sensitive applications.

Omega, follows a shared-state approach, where multiple concurrent schedulers can view the whole cluster state, with conflicts being resolved through a transactional mechanism. Omega schedulers use optimistic concurrency control to manipulate a shared representation of desired and observed cell state stored in a central persistent store. Kubernetes exploits Docker containers to map applications onto multiple host nodes and it is the open source version of Omega and Borg. Contrary to Mesos, Borg centralizes the resource management using a request-based mechanism, using priorities and admission quotas for scaling purposes. It offers a "one-size-fits-all" Remote Procedure Call (RPC) interface, state machine semantics, and a scheduler policy. Similarly to our flexible system, they work in coarse-grained mode but, they do not schedule applications but rather deployments. For this reason, only when all the resources are allocated, the user is able to launch her application.

Cluster management systems like HCloud [Delimitrou and Kozyrakis, 2016] or Quasar [Delimitrou and Kozyrakis, 2014] combine the ideas of previous works to provide a complete control over the cluster. Hcloud [Delimitrou and Kozyrakis, 2016] is a hybrid provisioning system that uses both reserved and on-demand resources. HCloud determines which jobs should be mapped to reserved versus on-demand resources based on overall load, and resource unpredictability. It also determines the optimal instance size an application needs to satisfy its constraints.

Quasar's [Delimitrou and Kozyrakis, 2014] main goal is to increase resource utilization. First, opposite to resource reservations, users express performance constraints for each workload, letting Quasar determine the right amount of resources to meet these constraints at any point. Then, it uses classification techniques to quickly determine the impact of the amount of resources on performance

for each workload and dataset. Finally, it uses this last result to jointly perform resource allocation and assignment.

Tarcil [Delimitrou et al., 2015] is a distributed scheduler that leverages information on the type of resources applications. It uses an analytically derived sampling framework that adjusts the sample size based on load, and provides statistical guarantees on the quality of allocated resources. It also implements admission control when sampling is unlikely to find suitable resources.

To identify the specific resources that are appropriate for incoming tasks, Paragon [Delimitrou and Kozyrakis, 2013] uses classification to determine the impact of platform heterogeneity and workload interference on an unknown, incoming workload. It assumes that the cluster manager has full control over all resources, which is often not the case in public clouds.

Condor [Tannenbaum et al., 2001] divides the nodes into two groups: the Submit nodes that are responsible for managing a queue of jobs and for all the aspects of the job's life cycle; and the Execution nodes that are responsible for the job execution and have their own policies for scheduling. A central manager serves as global rendez-vous point and it has global scheduling and matchmaking responsibilities.

Additionally, container orchestration frameworks, such as Docker Swarm[2], also provide efficient and scalable solutions to the problem of scheduling (that is, placing and provisioning) containers in a cluster.

Our work (presented in chapter 3) relies on many of the above systems, and can use them as a back-end to support scheduling of high-level applications rather than provisioning low-level containers. Existing auxiliary deployment tools such as Aurora[3] and Docker Compose[4], do not address scheduling problems.

Additionally, many works address the problem at a lower level and focus on task scheduling. Such schedulers are designed to support a specific "data-flow" programming model, but many of their design choices can also be used at a higher level. For example, Tyrex [Ghit and Epema, 2016] and HFSP [Dell'Amico et al., 2014; Pastorelli et al., 2013] are a sample of size-based schedulers, which is a family of policies known to drastically improve turnaround times, as we also have verified with our experiments.

In particular, Tyrex [Ghit and Epema, 2016] is a size-based resource allocation for MapReduce frameworks. Tyrex partitions the resources of a MapReduce framework, allowing any job running in any partition to read data stored on any machine, imposes runtime limits in the partitions and successively executes parts of jobs in a work-conserving way in these partitions until they can run to completion.

Similarly, Quincy [Isard et al., 2009] and DRF [Ghodsi et al., 2011] study max-min fair, task-level resource allocation, specifically working on multi-dimensional resources. Although our system currently consider a one-dimensional packing problem, due to the characteristics of the back-end we use, which does not yet support CPU-level partitioning, ideas presented in [Ghodsi et al., 2011] can be extended to our context, considering alternative back-ends supporting multi-resource partitioning.

[2] https://docs.docker.com/swarm/
[3] http://aurora.apache.org/
[4] https://docs.docker.com/compose/

Recently, schedulers supporting complex directed acyclic graphs representing low-level, parallel computations have also appeared: Graphene [Grandl et al., 2016], for example, addresses the problem of complex dependencies and heterogeneous demands among the various stages of the computational graph.

The work in [Rasley et al., 2016] indicate substantial improvements in terms of resource utilization (and not only allocation) thanks to worker queues, that independently schedule tasks. Bistro [Goder et al., 2015] employs a novel hierarchical model of data and computational resources, which enable efficient scheduling of data-parallel workloads. Firmament [Gog et al., 2016] is a centralized scheduler that has been shown to scale to over ten thousand machines at sub-second task placement latency, using a min-cost max-flow optimization approach.

Issues related to scheduling scalability, due to the sheer number of low-level tasks that are typically required by analytic jobs, have been addressed through a distributed design, such as in Sparrow [Ousterhout et al., 2013] and in Condor [Tannenbaum et al., 2001]. Although working at the application-level as we do in this thesis imposes a low toll on the scheduler, distributed designs are interesting also from the failure tolerance point of view.

Dynamic resource allocation

Dynamic resource allocation has been approached in many different ways in the literature [Ananthanarayanan et al., 2012; Babaioff et al., 2017; Curino et al., 2014; Delgado et al., 2016, 2015; Ghodsi et al., 2011; Grandl et al., 2014; Hassan and Zwaenepoel, 2017; Kuzmanovska et al., 2014, 2016; Lo et al., 2015; Padala et al., 2007; Rasley et al., 2016; Shahrad et al., 2017; Verma et al., 2015; Yang et al., 2017].

The authors in [Kuzmanovska et al., 2014, 2016] present a solution called KOALA-F that is based on a feedback control loop which requires every *framework* running inside the cluster to periodically send information to the scheduler. Every framework must be enhanced so that it can talk directly to KOALA-F, in order to transmit information about their metrics in form of color: red, yellow and green. KOALA-F will allocate or deallocate resources to that specific framework based on that color. Red means that the framework is struggling due to lack of resources, yellow is the good state, while green means that there are more resources than needed. In contrast, our work (presented in chapter 4) does not require such instrumentation; we are completely agnostic to the application that is running and we use general metrics in order to dynamically reallocate the resources of the running applications. In addition, we operate on the application rather than on the framework level.

The authors in [Curino et al., 2014] introduce a type of scheduler that is reservation-based. They propose a reservation definition language (RDL) that allows users to declaratively reserve access to cluster resources. They formalize the planning of current and future cluster resources as a Mixed-Integer Linear Programming (MILP) problem and they integrate their work in YARN [Vavilapalli et al., 2013]. In our dynamic scheduler, we avoid delegating this task to users by asking them to specify such information; most of the time the users have no knowledge of how their applications will behave.

The authors in [Padala et al., 2007] develop a feedback control loop for Virtual Machines (VMs), using a simple regression model to forecast the future allocation. They show that it is possible to reduce the gap in the CPU resource slack. However, unlike our work (see chapter 4), they do not address Memory resources and the consequences that under-provisioning has on applications, as we do in dynamic scheduler.

The authors in [Boutin et al., 2014] adopt a distributed scheduling architecture, whereby each scheduler aims at minimizing task completion time by careful placement strategies that use estimates of task runtime and their resource utilization. Contrary to our work, they use over-provisioning of resources and they tackle conflicts in an optimistic-manner. Our approach cooperates with an existing scheduler, instead of replacing it, and does not use task runtime to adjust cluster resources allocated to applications.

Some other works [Babaioff et al., 2017; Shahrad et al., 2017] propose to address the problem with economics principles. In particular, in [Shahrad et al., 2017] the authors build a pricing model that enables infrastructure providers to incentivize their tenants to use graceful degradation, a self-adaptation technique originally designed for constructing robust services that survive resource shortages. The authors in [Babaioff et al., 2017], present a framework for scheduling and pricing cloud resources, aimed at increasing the efficiency of cloud resources usage by allocating resources according to economic principles. However, they achieve that by allocating more capacity than what is physically available, i.e., over-provisioning, which is a solution prone to uncontrolled failures[5] when utilization exceeds available resources.

Finally, works such as [Ananthanarayanan et al., 2012; Delgado et al., 2016, 2015; Ghodsi et al., 2011; Grandl et al., 2014; Karanasos et al., 2015; Lo et al., 2015; Mao et al., 2016; Rasley et al., 2016], focus either on resource placement or on meeting Service Level Objective (SLO) requirements. In the first case they relate to a packing problem and try to optimize it; Karanasos et al Karanasos et al. [2015] suggest to dynamically re-balance the load across hosts if the packing performed at a certain time leads to uneven loaded hosts. In the second case they leverage the elasticity of some frameworks and they increase resources for applications that are falling behind on their SLO. Our work is orthogonal to such methods and can leverage them to improve the system performance.

The authors in Zhang et al. [2016] propose task scheduling and data placement techniques that rely on historical resource utilization. Specifically, they process the history of CPU utilizations using the Fast Fourier Transform (FFT). Leveraging the k-Means algorithm, they cluster patterns in three categories: periodic, constant and unpredictable. They exploit the patterns of periodic and constant categories to improve the quality of task scheduling.

Albeit all these works are valid and propose their own vision of the problem, they share one element: although some of them address a multi-dimensional packing problem for provisioning resources to applications, when it comes to reclaiming resources granted to applications they mostly focus on “time

[5]The Operating System (OS) kills processes due to Out Of Memory (OOM) following its own algorithm.

sharable" resources, like the CPU, rather than "finite" resources like Memory.[6] As a consequence, such methods are limited to improve system efficiency from the perspective of CPU utilization.

An example of prior work that modulates "finite" resources is Borg [Verma et al., 2015]. Borg features a resource reclamation system that seizes unused resources and offers them to other applications. The authors study the impact of wrong memory reallocation on running tasks, which causes resource contention: the OS enters a special state to kill processes that are OOM. The authors present different levels of "rigidity" for their reclamation system (baseline, medium and aggressive) and show both the benefit and the number of OOMs events for each of them. They conclude by accepting the trade-off obtained by the medium setting. Instead, we present a dynamic allocation system that relies on online resource forecasting, with accurate quantification of uncertainty. In addition, we seek to gain control over the OS and minimize application failures events while maximizing the resource utilization.

[6] On the one hand, a resource is considered "time sharable" when the OS is able to use time sharing for scheduling it, and thus it does not impose limits on its availability. On the other hand, "finite" resources are those that cannot be sliced in time and thus cannot be effectively shared by multiple processes.

2.2 Time-Series Analysis

In this section we introduce time-series analysis and some methodologies that can be used in order to forecast values. These methodologies are used in chapter 4 to improve both cluster resource utilization and system responsiveness.

A time-series (fig. 2.1) is a sequential set of data points, measured typically over successive times. It is mathematically defined as a set of vectors $x(t), t = 0, 1, 2, ...$ where t represents the time elapsed [Adhikari and Agrawal, 2013; Brockwell and Davis, 2016]. The variable $x(t)$ is treated as a random variable. The measurements taken during an event in a time series are arranged in a proper chronological order.

A time-series containing records of a single variable is termed as univariate. But if records of more than one variable are considered, it is termed as multivariate. A time series can be continuous or discrete. In a continuous time series observations are measured at every instance of time, whereas a discrete time series contains observations measured at discrete points of time; in this thesis we use the latter. Usually in a discrete time series the consecutive observations are recorded at equally spaced time intervals.

A time-series in general is supposed to be affected by four main components, which can be separated from the observed data. These components are: *Trend*, *Cyclical*, *Seasonal* and *Irregular* components.

Trend The general tendency of a time series to increase, decrease or stagnate over a long period of time is termed as Secular Trend or simply Trend. Thus, it can be said that trend is a long term movement in a time series.

Cyclical The cyclical variation in a time series describes the medium-term changes in the series, caused by circumstances, which repeat in cycles. The duration of a cycle extends over longer period of time.

Seasonal Seasonal variations in a time series are fluctuations within a time-frame during the season.

Irregular Irregular or random variations in a time series are caused by unpredictable influences, which are not regular and also do not repeat in a particular pattern.

In our context, time-series are obtained by measuring at discrete points of time resource utilization (CPU, Memory, I/O) of a scheduled job. Since the life-span of a job inside the system may vary from several minutes to several days, not all the time-series will show all four characteristics. For example, seasonality is usually not present. We will see that some methodologies to forecast time-series values can deal with unexpected and missing characteristics, while other cannot.

In time series forecasting, past observations are collected and analyzed to develop a suitable mathematical model which captures the underlying data generating process for the series. The future events are then predicted using the model. This approach is particularly useful when there is not much

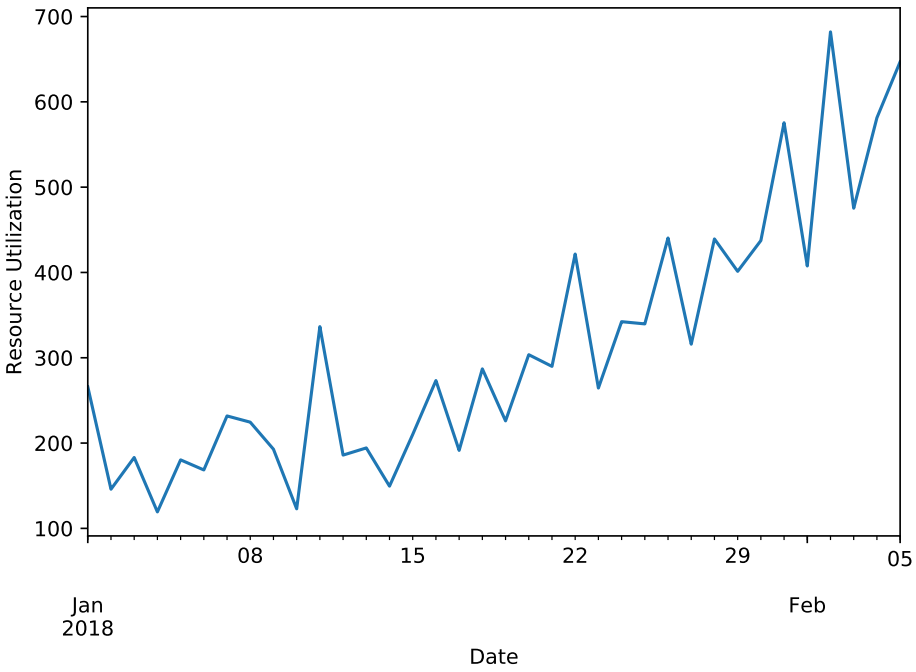


Fig. 2.1 Time-Series Example.

knowledge about the statistical pattern followed by the successive observations or when there is a lack of a satisfactory explanatory model.

The procedure of fitting a time-series to a proper model is termed as Time Series Analysis [Brockwell and Davis, 2016]. In practice a suitable model is fitted to a given time series and the corresponding parameters are estimated using the known data values. It comprises methods that attempt to understand the nature of the series and is often useful for future forecasting and simulation. One such method is based on regression analysis.

The goal of regression is to estimate relationships between the variables; many techniques for carrying it out have been developed. We can divide them in two distinct groups; parametric and non-parametric models. The first assume that the regression function is defined in terms of a finite number of unknown parameters (estimated from the observed data), while the second allow the regression function to lie in a specified set of functions which my have an infinite dimension space.

In our context, we analyzed two techniques to forecast time-series values: a parametric (ARIMA) and a non-parametric (GP regression). We will see that using a non-parametric approach have some benefits over a parametric.

2.2.1 Autoregressive Integrated Moving Average (ARIMA)

ARIMA is often considered as the “go-to method” for time series forecasting: it is a generalization of the Autoregressive Moving Average (ARMA) model to cope with non-stationary time series data, which

appear frequently in real-life applications such as the one we consider in this dissertation. Considering observation y_t at time t, the ARMA(p',q) model is described as follows:

$$y_t - \alpha_1 y_{t-1} - ... - \alpha_{p'} y_{t-p'} = \varepsilon_t + \theta_1 \varepsilon_{t-1} + ... + \theta_1 \varepsilon_{t-q} \tag{2.1}$$

where α are the parameters of the autoregressive part of the model, the θ are the parameters of the moving average part and the ε are error terms. In particular, p' and q are integers greater than or equal to zero and refers to the order of the autoregressive and moving average parts of the model respectively.

The underlying idea of ARIMA is that current values of a time series can be obtained by a linear combination of its past values, using finite differencing to produce stationary data. Formally, the ARIMA(p,d,q) model using lag polynomials is given below:

$$(1 - \sum_{i=1}^{p} \phi_i L^i)(1-L)^d y_t = \delta + (1 + \sum_{I=1}^{q} \theta_i L^i)\varepsilon_t \tag{2.2}$$

where $p = p' - d$, δ is a constant and L is defined as the lag or back-shift operator. d is an integer greater than or equal to zero and refer to the order of the integrated parts of the model and controls the level of differencing. Generally $d = 1$ is enough in most cases. An in-depth discussion about ARIMA can be found in [Brockwell and Davis, 2002].

Note that the ARIMA model is only effective for short-term forecasting. The forecasting power of an ARIMA model decreases as the forecasting horizon increases because the ARIMA forecast converges to the mean of the observations as the forecast horizon grows [Sumway and Stoffer, 2006]. In our case, the predictions that we make are short-term, thus we will not be affected by this limitation of ARIMA.

One important limitation of ARIMA is that we must know beforehand the characteristics of a time-series in order to correctly steer the model parameters p, d and q. Some implementation of ARIMA (e.g., in the R programming language), are able to infer these parameters automatically with a given time-series. However in our context, this is not always possible because the characteristics of our time-series may change in time as we will see in Chapter 4. Model selection, that is, searching through combinations of order parameters to pick the set that optimizes model fit criteria, is carried out using the Akaike information criteria, a method that is widely available in most ARIMA implementations. Note that parameter optimization is an operation that needs to be performed multiple times during a forecasting period, to adapt to variations in the time series characteristics.

Finally, most ARIMA implementations output confidence intervals associated with the selected model parameters [Brockwell and Davis, 2002]. We note that confidence intervals should not be confused with prediction intervals: the former are associated to the probability of the true model parameters to be within the confidence interval, whereas the latter are associated to the likely range of future values output by the model. As discussed in the literature [Brockwell and Davis, 2002], confidence intervals for the mean are generally much narrower than prediction intervals. This has a direct consequence in the context of this dissertation, which revolves around the idea of using predictive confidence to steer

system behavior: for this reason, in the next section, we present a Bayesian approach to time series modeling that features a principled approach to compute predictive confidence.

2.2.2 Gaussian Process Regression

In this section we review a non-parametric approach to regression. A non-parametric method does not make particular assumptions regarding the form of the function of interest. In practice, it allows us to fit any time-series without the need of having prior knowledge about the characteristics of the time-series itself. One additional benefit of GP regression is that the predicted output is not just a value, but a distribution which quantifies the uncertainty of the prediction.

In the GP literature, time series are treated as state space models, which are generalizations of auto-regressive models [Frigola-Alcalde, 2015; McHutchon, 2015]. Considering state x_t and observation y_t at time t, a state space model is described as follows:

$$\begin{aligned} x_{t+1} &= f(x_t) + \varepsilon_t \\ y_t &= g(x_t) + v_t \end{aligned} \tag{2.3}$$

where $f(x_t)$ is the state transition function and ε_t is the process noise, which follows a normal distribution. The state x_t may not be observed directly; an observation y_t is given as a function of the state $g(x_t)$, which is additionally corrupted by observation noise v_t.

According to Equation (2.3), a time series is modeled as a non-linear Markovian dynamical system. The Markov property implies that the current state x_t is *conditionally* independent from past states $\{x_\tau : \tau < t-1\}$, given the previous state x_{t-1}. The same is not true for the observations however. Thus, given a collection of noisy observations $\{y_\tau : \tau \leq t\}$, the goal for time series prediction is to infer the future state x_{t+1}. This requires learning the functions f and g, which involves placing a GP prior over f and g. However, the posterior over a non-linear dynamical system is not Gaussian, thus several approximation methods have been proposed in the literature [Frigola et al., 2007; Svensson et al., 2016; Turner et al., 2010].

In the context of recording resource utilization, we can make some simplifying assumptions. It is reasonable to assume that an observation y_t matches the state x_t. Of course, we have to acknowledge that resource utilization constantly fluctuates; these fluctuations however can be sufficiently explained by the noise term ε_t, which now accounts for both the process and the observation noise. We shall additionally make the dependency on past states explicit; for a history window of size h, we consider the following state-space model:

$$y_t = f(y_{t-1}, \ldots, y_{t-h}) + \varepsilon_t \tag{2.4}$$

To make predictions, we shall learn the transition function f by means of standard GP regression. From Equation (2.4), the transition function depends on the history explicitly. In this way, we avoid the additional costs of approximating the true posterior of a non-linear dynamical system.

A GP model transfers information across points that are considered similar, as this is reflected in the choice of kernel $k(x,x')$, which determines the prior covariance between inputs x and x'. If we assume that the inputs $\mathbf{X}$ solely consist of the recorded times, then similarity is only a matter of temporal locality, which is not optimal practice if the aim is to predict sudden changes of behavior throughout the course of a time series.

Hence, we resort to the definition of a kernel that relies on the observation history. It is implicitly assumed that if two sequences of observations are similar, then they must have been caused by the same "hidden" background processes; it is reasonable then to extrapolate and predict that the future observations will be similar as well. Such a history-dependent kernel can be easily constructed by transforming the data in an appropriate way. Consider a history window of size h, the training instances will be utilization patterns expressed as vectors of the form:

$$\tilde{\mathbf{x}}_t = [x_t, y_{t-h}, \ldots, y_{t-1}]^\top \tag{2.5}$$

where x_t is the t-th recorded time. Therefore, the history-dependent kernel is implemented by applying a typical exponential kernel on the transformed inputs:

$$k_h(x,x') = k(\tilde{\mathbf{x}}, \tilde{\mathbf{x}}') \tag{2.6}$$

Two different inputs x and x' will be similar if they have a similar history pattern, or equivalently, if the h preceded inputs have similar outputs. Note that we have kept the recorded times x_t along with the history, thus we do not completely ignore locality in the original input space.

2.3 Compute and Storage Disaggregation

To further improve the efficiency and responsiveness of distributed applications we study their performance in current cloud providers architectures. We are seeing a trend in which *(i)* providers tend to disaggregate storage and compute clusters[7] and *(ii)* new technologies (e.g., IBM Storlets [Rabinovici-Cohen et al., 2014]) are being developed to ship part of the code toward the storage, reducing the data transmitted to the compute [Moatti et al., 2017]. Private and public cloud providers are now able to give the user the possibility to deploy different types of frameworks that interact with the same object storage.

Indeed, thanks to virtualization, compute and storage clusters are more flexible, they can be easily provisioned in different sizes, and destroyed when not needed[8]. Increasingly, such storage and processing systems are exposed to users as *services*, deployed on either public or private cloud computing environments, rather than on bare-metal machines in private clusters. Many companies offer Analytics-as-a-Service (AaaS) clusters to run a variety of applications: Amazon Web Services (AWS) with Elastic MapReduce[9], DataBricks Cloud[10], Cloudera Cloud[11] and Google Cloud Hadoop[12] are noteworthy examples.

In cloud computing environments, the architecture of analytics clusters is the result of the composition of several services, consisting of three (logically separated) layers: the *Compute layer* refers to all cluster nodes that run the data processing application (e.g., a Spark application); the *Data layer* refers to any combination of storage services (e.g., HDFS [Shvachko et al., 2010] or OpenStack Swift [Arnold, 2014]); and the *Storage layer* that physically stores the data, including ephemeral disks, object and elastic block stores. Additionally, it is likely for the Data or Storage layers and the Compute layer to be on different racks or even data-centers: as a consequence, the traditional wisdom of "moving the computation to the data" (i.e., *data locality*) may be challenged. For example, consider Amazon S3[13]: data resides on a set of machines dedicated just to storage, breaking data locality.

In the remaining of this sections we first take a closer look at the different components that can build an analytics service, then we briefly introduce one technology that is widely used for storage solution among the analytics services providers; the cloud object storage (e.g., Amazon S3, Openstack Swift).

2.3.1 Analytics services components

Analytics services consist of three main logical components, as illustrated in fig. 2.2:

1. The Compute layer runs parallel processing frameworks that execute analytics applications (or jobs), e.g., Apache Spark. An example of compute layer service is Amazon Elastic MapReduce.

[7] https://databricks.com/blog/2017/05/31/top-5-reasons-for-choosing-s3-over-hdfs.html
[8] https://aws.amazon.com/application-hosting/benefits/
[9] https://aws.amazon.com/emr/
[10] Solution hosted on AWS: https://databricks.com/product/databricks-cloud
[11] Solution hosted on AWS: http://www.cloudera.com/content/cloudera/en/solutions/partner/Amazon-Web-Services.html
[12] https://cloud.google.com/hadoop/
[13] https://aws.amazon.com/s3/

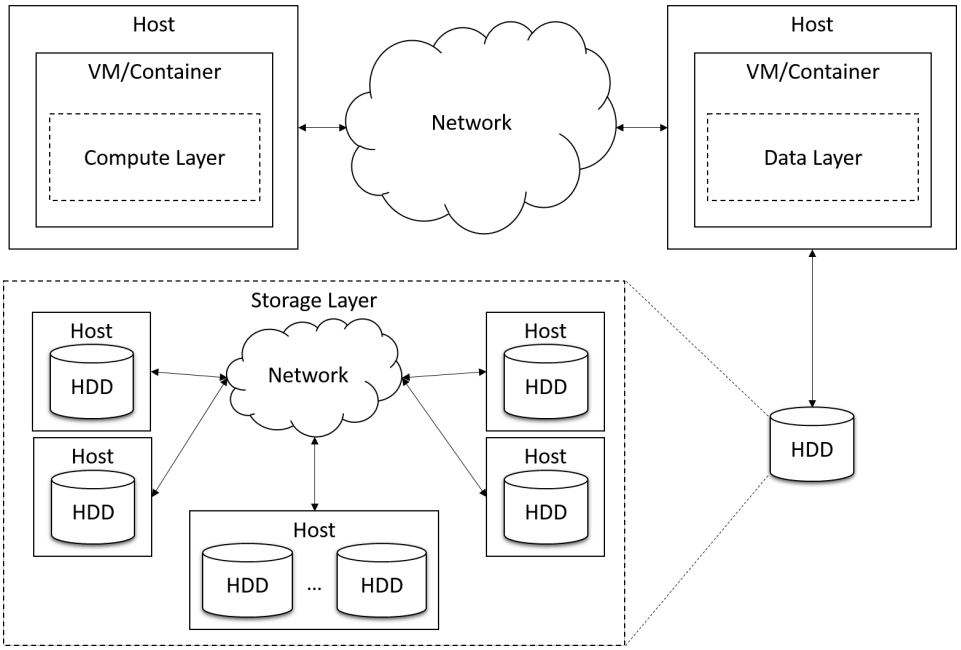


Fig. 2.2 Logical components in an analytics cluster.

2. The Data layer exposes logical storage services from the underlying physical storage system to the Compute layer. Examples of data layers include Hadoop HDFS and Amazon S3.

3. The Storage layer handles read/write operations and has direct access to the data. Disks exposed to the data layer can be a series of individual, ephemeral devices as well as more complex distributed file systems.

The distinction between Data and Storage layers is a consequence of virtualization. Indeed, a VM hosting the Data layer (e.g., the HDFS' DataNode) might use a virtual disk which resides on a different host. For example, many public cloud providers expose virtual disks provisioned by an underlying distributed file system to improve, among others, VMs' migration time. In general, the Compute layer requires a compatible Data layer to access the Storage layer. Large-scale computing frameworks do not support all the distributed file systems. For example, when deploying a VM on AWS with EBS[14], Spark cannot be used because the distributed file system used by Amazon to deploy their EBS system is opaque to the virtual machines and thus to Spark. Therefore there is the need of introducing a compatible Data layer (e.g., HDFS).

Another important concept are ephemeral disks; they are a temporary storage that can be implemented, by the cloud infrastructure administrators, in different ways. One solution could be to have such ephemeral disk directly connected to a portion of a single physical disk that reside on the same host that runs the VM/container. A second solution could be to connect the ephemeral disk to a different

[14] https://aws.amazon.com/ebs/

distributed file system, much like the configuration with volumes that we described in the previous paragraph.

2.3.2 Cloud Object Storage

Object storage is one of three types of cloud storage that are currently wide spread, the other two are: File and Block storage. Applications developed in the cloud often take advantage of object storage's scalability and metadata characteristics.

An object encapsulates data and metadata describing the object and its data. Additionally, the entire object is created at once and cannot be updated in place, although the entire value of an object can be replaced. This simple object semantics enables the implementation of highly scalable, distributed and durable storage that can provide very large capacities at low cost. Object storage is typically accessed through RESTful HTTP, which is a good fit for cloud applications. It is ideal for storing unstructured data, e.g., video, images, backups and documents such as web pages and blogs. Examples of object storage systems include Amazon S3, Azure Blob storage[15], OpenStack Swift and IBM Cloud Object Storage[16].

Object storage has a shallow hierarchy. A storage account may contain one or more buckets or containers (hereafter we use the term container), where each container may contain many objects. Typically there is no hierarchy in a container, e.g., no containers within a container, although there is support for hierarchical naming. This is different than file systems where there is both hierarchy in the implementation as well as in naming.

Common operations on object storage include:

- *PUT Object* which creates an object, with the name, data and metadata provided with the operation
- *GET object* which returns the data and metadata of an object
- *HEAD Object* which returns just the metadata of an object
- *DELETE Object* which deletes an object
- *GET Container* which lists the objects in a container
- *HEAD Container* which returns the metadata of a container

Object creation is atomic, so that two simultaneous PUTs on the same name will create an object with the data of one PUT, but not some combination of the two. Object storage does not have an atomic rename operation; rename can be emulated non-atomically through COPY and DELETE.

In order to enable a highly distributed implementation the consistency semantics for object storage often includes some degree of *eventual consistency* [Vogels, 2009]. Eventual consistency guarantees that if no new updates are made to a given data item, then eventually all accesses to that item will return

[15] https://azure.microsoft.com/en-us/services/storage/blobs/
[16] https://www.ibm.com/cloud-computing/products/storage/object-storage/cloud/

the same value. There are various degrees of eventual consistency. For example, AWS guarantees read after write consistency for its S3 object storage system, i.e., that a newly created object will be instantly visible. Note that this does not necessarily include read after update, i.e., that a new value for an existing object name will be instantly visible, or read after delete, i.e., that a delete will make an object instantly invisible. An important aspect typical to most object stores concerns the listing of the objects in a container; the creation and deletion of an object may be eventually consistent with respect to the listing of its container. In particular, a container listing may not include a recently created object and may not exclude a recently deleted object.

2.3.3 Related Work on Data Locality and Cloud Object Stores Performance

The performance implications of data locality, a consequence of disaggregated compute and storage, have been investigated in several studies. We can identify two major trends: one (e.g., [Ananthanarayanan et al., 2011; Nightingale et al., 2012; Ousterhout et al., 2015]) arguing that data locality is not relevant, while the other (e.g., [Guo et al., 2012; Ranganathan and Foster, 2002; Roman et al., 2015; Venzano and Michiardi, 2013; Wang et al., 2009; Xie et al., 2010]) highlighting the opposite. All these works, albeit valid, base their conclusion on limited information and define data locality as a boolean feature (present or not). We move on from this dichotomy by investigating applications' performance in a variety of service compositions, leading to various degrees of data locality.

Considering the methodology, we can divide the recent research efforts to understand the impacts of data locality into three main categories: *(a)* analysis on limited/public configurations, *(b)* analysis on limited workloads and *(c)* theoretical or trace-driven analysis.

Limited/Public Configurations Examples of works that use limited/public deployments can be found in [Ananthanarayanan et al., 2011; Guo et al., 2012; Nightingale et al., 2012; Ousterhout et al., 2015; Xie et al., 2010]. For example, Ousterhout et al. in [Ousterhout et al., 2015] use an ideal scenario in terms of data locality (Compute and Data layer on the same VM), with limited knowledge of the underlying Storage layer. With the help of an analysis performed on network, disk block time and percentages of resource utilization, such work state that the runtime of analytics applications is generally CPU-bound rather than I/O intensive; thus, data locality may be considered irrelevant. A recent work [Trivedi et al., 2016] shows that this is not always true; moving from a 1Gbps to a 10Gbps network can have a huge impact on the application runtime.

Limited Workloads The studies presented in [Roman et al., 2015; Xie et al., 2010], and [Rupprecht et al., 2014] use a limited set of workloads to investigate data locality. For example, Xie et al. in [Xie et al., 2010] use two workloads: a WordCount and a Grep-like applications to demonstrate that data placement plays an important role in analytics applications. While this consideration is valid, with our approach, we recognize the importance of workload heterogeneity in studying system performance.

Theoretical or Trace-driven Analysis Some authors base their works on theoretical or trace-driven analysis [Ananthanarayanan et al., 2011; Guo et al., 2012; Ranganathan and Foster, 2002; Wang et al., 2009]. The work from Ananthanarayanan et al. in [Ananthanarayanan et al., 2011] is based on Facebook's traces. The authors state that, since network technology evolves quickly, data locality is an aspect that will soon be neglected; they also use micro benchmarks to study a single aspect of an analytics cluster. Instead, we show that in some cases the network might indeed not be a bottleneck, while in others it may contribute to harm application performance. In addition, using a micro-benchmark approach alone to measure I/O performance, can lead to inaccurate results, since analytics frameworks like Spark or Hadoop are more complex than a set of read/write operations.

Other Limitations Works like [Lin et al., 2012; Yang and Sun, 2011; Zhang et al., 2006] do not fall directly into one of these categories, as they model different aspects of a MapReduce application. Nonetheless, they leave data locality as an abstract concept and they always consider configurations when Compute, Data and Storage are on the same host. For example, Yang et al. in [Yang and Sun, 2011] model the relationship between number of mapper and reducers, while Lin et al. in [Lin et al., 2012] model an entire analytics application that uses Hadoop and an analytics framework. Zhang et al. in [Zhang et al., 2006] create a model to improve the data locality when Compute, Data and Storage layers are on the same machine. We go further and study data locality when there is a clear separation of Compute and Data layer.

Albeit valid, all the previous works address a specific storage type, that is file storage. Object storage has not been the focus since it is not native to the ecosystems of analytics frameworks. Our work and works from [Rupprecht et al., 2014, 2017] show the existence of an impedance mismatch between the analytics frameworks and object storage that imposes a toll on performance. Moreover, we show that it is possible to improve performance by eliminating the impedance mismatch between the compute and storage layer, which can highly affect the run times of such applications, in particular, when using object storage (e.g., Openstack Swift [Arnold, 2014]). There has also been some work from industry and open source to improve this impedance mismatch. Databricks introduced the DirectOutputCommitter[17] for Amazon S3, but it failed to preserve the fault tolerance and speculation properties of the temporary file/rename paradigm. At the same time Hadoop developed version 2 of the FileOutputCommitter[18], which renames files when tasks complete instead of waiting for the completion (commit) of the entire job. However, this solution does not solve the entire problem.

Current connectors from the Hadoop community for the OpenStack Swift and Amazon S3 APIs [19], can also lead to failures and incorrect executions due to eventual consistency. Some work has been done

[17] https://github.com/apache/spark/pull/12229
[18] https://issues.apache.org/jira/browse/MAPREDUCE-6336
[19] https://aws.amazon.com/sdk-for-java/

to address this problem. EMRFS[20][21] from Amazon and S3mper[22][23] from Netflix overcome eventual consistency by storing file metadata in DynamoDB [Sivasubramanian, 2012], an additional storage system separate from the object storage that is strongly consistent. A similar feature called S3Guard[24][25] is being developed by the Hadoop open source community for the S3a connector. Solutions such as these that require multiple storage systems are complex and can introduce issues of consistency between the stores. They also add cost since users must pay for the additional strongly consistent storage. Our solution does not require any extra storage system.

Recently Databricks introduced a new commit protocol called DBIO[26] that removes the impedance mismatch and guarantees fault tolerance. This new transactional commit protocol provides strong guarantees in the face of various types of failures. Moreover, by enforcing correctness, it is able to provide safe task speculation, atomic file overwrite and consistency for Spark output. DBIO achieves similar objectives to the solution that we present in Chapter 6, but the solution is proprietary, whereas we fully describe our work, put it in open source, and thoroughly analyze its performance.

[20] https://aws.amazon.com/blogs/aws/emr-consistent-file-system/
[21] http://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-fs.html
[22] http://techblog.netflix.com/2014/01/s3mper-consistency-in-cloud.html
[23] https://github.com/Netflix/s3mper
[24] https://issues.apache.org/jira/browse/HADOOP-13345
[25] http://www.slideshare.net/hortonworks/s3guard-whats-in-your-consistency-model
[26] https://databricks.com/blog/2017/05/31/transactional-writes-cloud-storage.html

Chapter 3

A Flexible Scheduling Heuristic

The last decade has witnessed the proliferation of numerous distributed frameworks to address a variety of large-scale data analytics and processing tasks. First, MapReduce [Dean and Ghemawat, 2008] has been introduced to facilitate the processing of bulk data. Subsequently, more flexible tools, such as Dryad [Isard et al., 2007], Spark [Zaharia et al., 2012], Flink[1] and Naiad [Murray et al., 2013], to name a few, have been conceived to address the limitations and rigidity of the MapReduce programming model. Similarly, specialized libraries such as MLLib [Meng et al., 2016] and systems like TensorFlow [Abadi et al., 2016] have seen the light to cope with large-scale machine learning problems. In addition to a fast growing ecosystem, individual frameworks are driven by a fast-pace development model, with new releases every few months, introducing substantial performance improvements. Since each framework addresses specific needs, a wide choice of tools and combinations thereof are available to users that now can address the various stages of their data analytics projects. To the best of our knowledge, no existing tool currently addresses the problem of scheduling analytic applications as a whole, leveraging the intrinsic properties of the frameworks such applications use, but without requiring substantial modification of such frameworks.

The endeavor of this chapter is to fill the gap that exists in current approaches, and *raise the level of abstraction* at which scheduling works. We introduce a general and flexible definition of *applications*, how they are composed, and how to execute them. For example, a user application addressing the training of a statistical model involves: a user-defined program implementing a learning algorithm, a framework (e.g., Spark) to execute such a program together with information about its resource requirements, the location for input and output data and possibly parameters exposed as application arguments. Users should be able to express, in a simple way, how such an application must be packaged and executed, submit it, and expect results as soon as possible. We show that scheduling such applications represents a departure from what has been studied in the scheduling literature, and we present the design of a new algorithm to address the problem. A key insight of our approach is to exploit the properties of the frameworks used by an application, and distinguish their components according to classes, core and

[1] https://flink.apache.org/

elastic: the first being required for an application to produce work, the latter contributing to reduced execution times.

Our heuristic focuses cluster resources to few applications, and uses the class of application components to pack them efficiently. Our scheduler aims at high cluster allocation and a responsive system. It can easily accommodate a variety of scheduling policies, beyond the traditional "first-come-first-served" or "processor sharing" strategies, that are currently used by most existing approaches. We study the performance of our scheduler using realistic, large-scale workload traces from Google[2] [Reiss et al., 2012, 2011; Wilkes, 2011], and show it consistently outperforms the existing baseline approach which ignores component classes: application turnaround times are more than halved, and queuing times are drastically reduced. This induces fewer applications waiting to be served, and increases resource allocation up to 20% more than the baseline. Finally, we build a full-fledged system, called Zoe, that schedules analytic applications according to our original algorithm and that can use sophisticated policies to determine application priorities. Our system exposes a simple and extensible configuration language that allows application definition. We validate our system with real-life experiments, and report conspicuous improvements when compared to a baseline scheduler, when using a representative workload: median turnaround times are reduced by up to 37% and median resource allocation is 20% higher.

This chapter is organized as follows. We start by clarifying what analytic applications are, give examples and formulate our problem statement in Section 3.1. We then describe the details of our flexible scheduling heuristic, in Section 3.2, which we evaluate using simulations in Section 3.3. Finally, the system implementation is described in Section 3.4, and its evaluation is presented in Section 3.5.

3.1 Definitions and Problem statement

3.1.1 Definitions

We define a data analytics **framework** as a set of one or more software **components** (executable binaries) to accomplish some data processing tasks. Distributed frameworks are generally composed by a controller, a master and a number of worker components. Examples of distributed frameworks are Apache Spark[3], Google TensorFlow[4] and MPI[5]. Another example of a simple data analytics framework we consider is an interactive Notebook [Ragan-Kelley et al., 2014].

Distributed frameworks require a *scheduler* to orchestrate their work: they execute *jobs*, each of which consists of one or more *tasks* that run in parallel the same program. Such schedulers operate at the *task level*: they assign tasks to workers, and they are highly specialized to take into account the peculiarities of each framework.

[2] https://github.com/google/cluster-data
[3] http://spark.apache.org/
[4] https://www.tensorflow.org/
[5] https://www.open-mpi.org/

Framework schedulers such as Mesos [Hindman et al., 2011] and Yarn [Vavilapalli et al., 2013] introduce an additional scheduling component to share cluster resources among concurrent frameworks: sharing policies are based on simple variations of Processor Sharing. Similarly, *cluster management systems* such as Docker Swarm[6] and Kubernetes[7] use a scheduler that assigns resources to generic frameworks. The problem to solve is the *efficient allocation* of resources by placing framework components and their tasks on cluster machines that satisfy a set of constraints.

We are now ready to define **analytics applications**, which are the elements we schedule in this chapter. Our main objective is to *raise the level of abstraction* by manipulating an abstract entity encompassing one or more analytics frameworks, their components and the necessary logic for them to cooperate toward producing useful work by running *user-defined jobs*. What sets apart our work from the state of the art is that our scheduler takes into account the notion of **component classes**, which allows modeling the specificity of each framework. We have found two distinct component classes to be sufficient to model existing analytic frameworks: thus, framework components either belong to a **core** or to an **elastic** class. Core components are compulsory for a framework to produce useful work; elastic components, instead, optionally contribute to a job, e.g. by decreasing its runtime. Consider, for example, Spark. To produce work, it needs some core components: a controller (the spark client running the DAG scheduler), a master (in a standalone deployment), and one worker (running executors). We treat additional workers as elastic components. An alternative example is an application using TensorFlow, which only works with core components: one or more parameter servers and a number of workers. These two frameworks have substantially different runtime behavior: Spark is an elastic framework that can dynamically integrate workers to dispatch tasks. TensorFlow is rigid, and uses only core components to make progress.

To summarize, the nature of an application is that of raising the level of abstraction and an application is considered as being a collection of frameworks and their heterogeneous components as a single entity to schedule and allocate in a cluster of computers.

3.1.2 Problem Statement

We now treat the applications defined above as abstract entities that we call *requests*: they include one or more *components*, which belong to a given class, either core or elastic. In the literature, the classical problem of scheduling generic requests to be served by a distributed system has been extensively studied [Dutot et al., 2004; Pruhs et al., 2004; Sgall, 2015]. Requests composed solely by core components are usually referred to as *rigid*, while requests composed solely by elastic components are referred to as *moldable* (if the assigned resources are decided when the request is served and they do not change for the whole execution) or *malleable* (if the resources can vary during the execution[8]). A key difference

[6] https://docs.docker.com/swarm/

[7] http://kubernetes.io/

[8] An example of *malleable* framework is Spark[Zaharia et al., 2010]. Worker can be added or removed without destroying the application execution.

with respect to previous work is that we consider *heterogeneous requests*, composed by both core and elastic components.

For simplicity of exposition, we assume system resources that can be measured in units, and that there are R available units overall to satisfy the requests. Each request i specifies the amount of units for its core and elastic components, labeled C_i and E_i respectively. Ideally, with enough available resources, a request is allocated all of its components: in this case, we define the service (or execution) time as T_i. The amount of work to satisfy a request is the area of the square $W_i = T_i \times (C_i + E_i)$. More generally, a request is allocated *at least* $C_i + x_i(t)$ resources, where $0 \leq x_i(t) \leq E_i$. Then, the service time is $T_i' = \frac{W_i}{C_i + x_i(t)}$. This simple model allows updating the service time T_i' when a scheduling decision modifies $x_i(t)$, by measuring the amount of work accomplished so far, and by computing the remaining amount of work to be done. While more complex models to describe T_i' can be conceived, for example taking into account the multi-dimensional nature of system resources or different scalability models, our simple approximation doesn't affect the nature of the scheduling problem we are studying. In what follows we assume that each request can fully utilize the specified number of core and elastic components, if they are granted resources.

Essentially, the problem of scheduling the execution of an *incoming workload* of requests amounts to: *i)* sorting requests to decide in which order to serve them; *ii)* allocating distributed resources to requests selected for service. The sorting phase can be solved using naive approaches, e.g. FIFO ordering, or more sophisticated policies, that use request size information. Even more generally, requests can be placed into "pools" and be assigned priorities, to mimic the hierarchical organization of the users, for example. The allocation phase is more tricky: in the abstract, it is a "packing" problem that determines how to shape requests being served. Even assuming service times to be known a-priori (e.g., T_i is given as an input), it is well known that the *on-line scheduling problem* is NP-hard [Pruhs et al., 2004]. Therefore, we need to find a suitable heuristic to approximate a solution to the scheduling optimization problem. In our case, this amounts to minimizing the application *turnaround times*, which is the interval of time between request i submission and its completion. In the context we consider, optimizing the average turnaround time represents a meaningful performance metric, as it caters system responsiveness.

Our scheduling problem does not directly take care of data locality constraints. As we saw in [Pace et al., 2017], recently cloud providers tend to disaggregate compute and storage layer at different levels: a compute and data node can reside on the same host, on different hosts or even on different data centers. Next, we motivate our problem with a simple illustrative example.

Illustrative example We consider a system with 10 available resource units, and four requests waiting to be served, as shown in Figure 3.1. Each request needs 3 units for the core components, and different units for the elastic components. For each request, $T_i = 10$. In this example we focus on the allocation phase only and we use the FIFO policy to sort the pending requests.

Given these requests, a traditional, rigid approach to scheduling – which does not make the distinction between component classes – assigns all required resources to each request. Since all requests need at least 5 units ($C_i + E_i \geq 5$), and since any pair of requests have an aggregated need that exceeds 10 units,

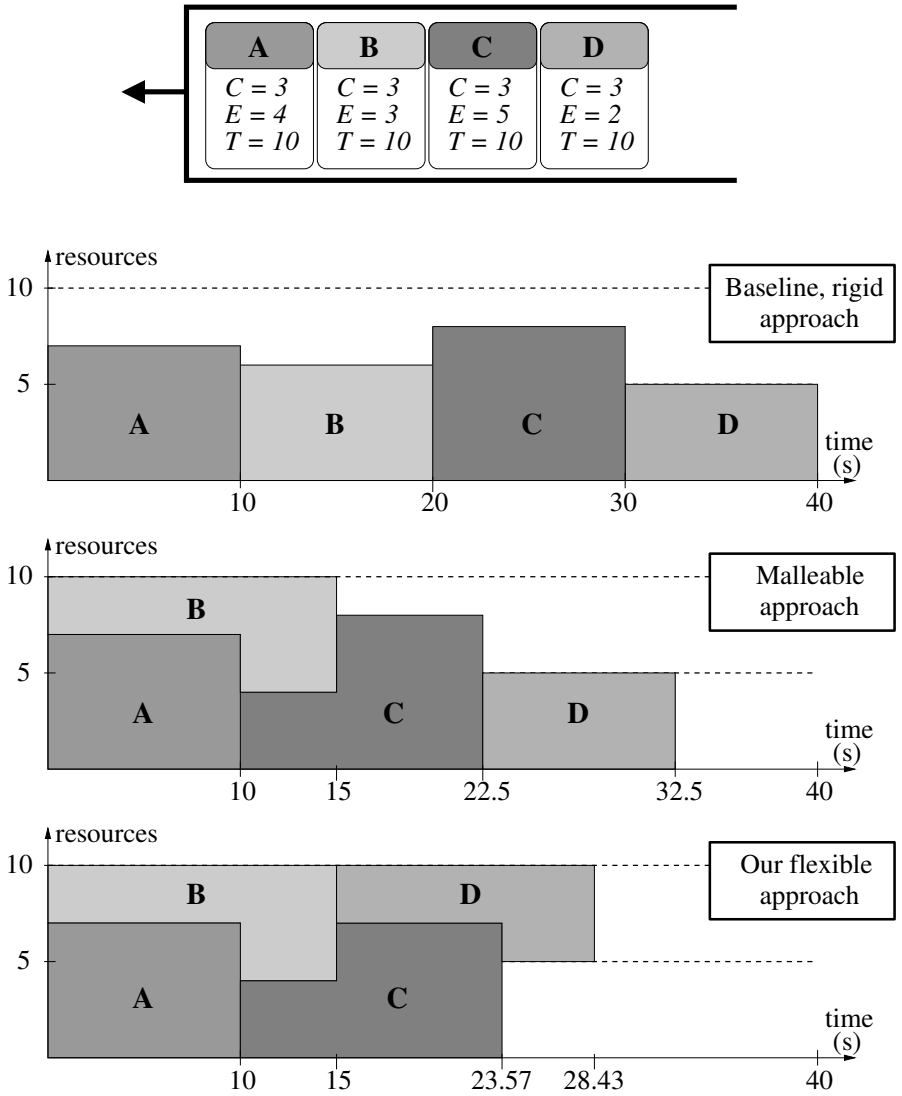


Fig. 3.1 Illustrative examples of request scheduling: (top) rigid, (middle) malleable, (bottom) flexible approaches.

the scheduler serves one request at a time (Figure 3.1, top): the average turnaround time is 25s. Note that, in this case, backfilling is not possible, i.e., even by changing the order in which requests are served the situation does not change.

Another scheduling approach comes from the literature of malleable job scheduling. The scheduler assigns all resources to the first request in the waiting line, then assigns the remaining resources (if any) to the next request, and so on, until no more free resources are available. This heuristic has been shown to be close to optimal [Dutot et al., 2004]. Figure 3.1, middle, illustrates the idea: request B can be served along with request A. When request A has completed, the scheduler first assigns more resources to request B, and then tries to serve the next request. Similarly, when request B has completed, the scheduler first assigns more resources to request C, then attempts at serving request D. However,

since request D needs at least $C_i = 3$ units, the scheduler is blocked (note that request C uses 8 units), so request D needs to wait, and some system resources remain unused. The average turnaround time is 20s.

We advocate the need for a new approach to scheduling, which distinguishes component classes. The idea is to exploit the flexibility of elastic components and use system resources more efficiently. Intuitively, a solution to the problems of existing heuristics is to reclaim some resources assigned to elastic components of a running request and assign them to a pending request. This is shown in the bottom of Figure 3.1: the scheduler reclaims just one unit from request C so that it can provide 3 units to request D, which are sufficient for starting its core components and produce useful work. With this approach, the average turnaround is 19.25s.

While the above solution seems simple, it poses many challenges: how many units assigned to elastic components can be sacrificed for serving the next request? How many requests should be served concurrently? Should the scheduler focus on core components alone, to make sure many requests are served concurrently? How can scheduling take into account the priorities assigned by the sorting phase?

The last point introduces an additional challenge, related to *preemptive scheduling* policies. If a high priority request arrives, since it is not possible to interrupt core components – for this would kill the request – how can we select and preempt elastic components to accommodate the new request?

Given heterogeneous, composite requests, which are neither rigid, nor malleable (but both), available scheduling heuristics in the literature fall short in addressing the sorting and allocation problems: a new approach is thus truly desirable.

3.2 A Flexible Scheduling Algorithm

3.2.1 Design guidelines

We characterize a request by its arrival time, its priority (to decide the order in which the requests should be served), the resources it asks for (core and elastic) and the execution time (in isolation, *i.e.,* when all required resources are granted to the application). Given an incoming workload, our goal is to optimize the sum of the turnaround times τ_i, that is:

$$\min \sum_i \tau_i \Rightarrow \min \sum_i (\text{queuing}_i + \text{execution}_i)$$

The actual execution time depends on the amount of resources assigned over time to the request. Now, recall that the scheduling problem can be broken into sorting and allocation phases. Sorting determines when a request is served, thus it has an impact on its queuing time. The allocation phase contributes both to queuing and actual execution times. Depending on allocation granularity [Schwarzkopf et al., 2013], a request might need to wait for a number of resources to be available before occupying them, thus increasing – albeit indirectly – the queuing time. The execution time is directly related to the allocation algorithm and to the workload characteristics.

We decouple request sorting from allocation:[9] our scheduler maintains the request ordering, as imposed by an external component, and only focuses on resource allocation. Sorting can be simply based on arrival times (which amounts to implement a FIFO queuing discipline), or can use additional information, such as request size (thus implementing a variety of size-based disciplines).

Overall, we optimize request turnaround times through careful resource allocation, and *design an algorithm that strives at allocating all available cluster resources, by serving the least number of requests at a time*. Intuitively, by "focusing" resources to few requests, we expect their execution times to be small. Consequently, queued requests also enjoy smaller wait times, because resources are freed more quickly.

3.2.2 Algorithm Details

Although we support preemptive scheduling policies, to simplify exposition, we first consider the case with no preemption: resources assigned to a request can only increase, and a new request can be placed, at most, at the head of the waiting line, depending on the sorting component. We stress that the output of our scheduling algorithm is a *virtual assignment*, *i.e.,* the mechanism to physically allocate resources according to the computed assignment (core and elastic components for running applications) is separate from the scheduling logic, and considered as an implementation detail.

Our resource allocation procedure is called REBALANCE, and it is triggered by two events: request arrivals and departures – see Algorithm 1. When a new request arrives (procedure ONREQUESTARRIVAL), the resource assignment is done only if such a request is placed at the head of the waiting line and there are unused resources that are sufficient for running its core components. When a request is completed (procedure ONREQUESTDEPARTURE), the released resources are always reassigned.

The scheduler maintains two ordered sets: the requests waiting to be served ($\mathscr{L}$), and the requests in service ($\mathscr{S}$). Each request *req* needs *req*.*C* core components and *req*.*E* elastic components; depending on the allocation, request *req* is granted $0 \leq req.G \leq req.E$ elastic components. The core of the procedure REBALANCE (lines 18-21) operates as follows: each request *req* in the serving set $\mathscr{S}$ has always **at least** *req*.*C* resources assigned. Excess resources are assigned to the requests in $\mathscr{S}$ following the request order. The scheduler assigns as many elastic components as possible to the first request, then to the second, and so on, in cascade.

Following the design guidelines, the set $\mathscr{S}$ should only contain the requests that are strictly necessary to use all the available resources. This is accomplished by the first part of the procedure REBALANCE (lines 8-13): a request is added to $\mathscr{S}$ if the current requests in $\mathscr{S}$ are not able to saturate the total resources ($total$, line 8). Note that we add a request to $\mathscr{S}$ only if there is room to allocate all of its core components.

[9] This approach is similar to the one used in the SLURM scheduler [Yoo et al., 2003], where the order of the pending jobs is given by an external, pluggable, component, and the scheduler processes the jobs following that order.

Algorithm 1: Resource assignment procedures (no preemption)

```
1  procedure OnRequestArrival(req)
2      Insert(req, 𝓛)
3      if req == 𝓛.head and req.C ≤ avail then
4          Rebalance()
5  procedure OnRequestDeparture()
6      Rebalance()
7  procedure Rebalance()
8      while Σ_{j∈𝓢} (req_j.C + req_j.E) < total and (𝓛 not ∅) do
9          req ← 𝓛.head
10         if req.C + Σ_{j∈𝓢} req_j.C < total then
11             Insert(pop(𝓛), 𝓢)
12         else
13             break
14     avail ← total − Σ_{j∈𝓢} req_j.C
15     forall req ∈ 𝓢 do
16         req.G ← 0
17     req ← 𝓢.head
18     while avail > 0 and (req not NULL) do
19         req.G ← min(req.E, avail)
20         avail ← (avail − req.G)
21         req ← req.next
```

Algorithm 2: Resource assignment procedures (with preemption)

```
1  procedure OnRequestArrival(req)
2      if req.P > 𝓢.tail.P then
3          if req.C ≤ Σ_{j∈𝓢} req_j.E then
4              Insert(req, 𝓢)
5              Rebalance()
6          else
7              Insert(req, 𝒲)
8      else
9          Insert(req, 𝓛)
10         if req == 𝓛.head and req.C ≤ avail then
11             Rebalance()
12 procedure OnRequestDeparture()
13     while 𝒲.head.C + Σ_{j∈𝓢} req_j.C < total and (𝒲 not ∅) do
14         Insert(pop(𝒲), 𝓢)
15     Rebalance()
```

3.2.3 Preemptive policies

We now consider preemptive policies: request arrivals can trigger (partial) preemption of running requests, e.g. if new requests have higher priority than that of the last request in service. In this case, the tuple describing a request also stores its priority, $req.P$. It is important to note that, in this work, the preemption mechanism only operates on elastic components of running applications, whereas core components (that are vital for an application) cannot be preempted.

Algorithm 2 shows the modifications to the procedures ONREQUESTARRIVAL and ONREQUESTDEPARTURE to support preemption. When a new request arrives, if its priority is higher than the requests in service, we check if its core components can be allocated using the resources occupied by the elastic components of currently running requests. If so, we insert the request into the set $\mathscr{S}$ and call REBALANCE (defined in Algorithm 1). Otherwise, we insert the request into an auxiliary waiting line $\mathscr{W}$, which is given priority when resources become available. Indeed, procedure ONREQUESTDEPARTURE indicates that we first consider the waiting requests in $\mathscr{W}$, and we add to the set $\mathscr{S}$ as many of them as possible, considering solely the core components. In other words, requests in $\mathscr{W}$ have higher priority than those in $\mathscr{L}$. Finally, the call of REBALANCE assigns the remaining resources to the elastic components of high priority requests.

3.3 Numerical evaluation

3.3.1 Methodology

We evaluate our algorithm using an event-based, trace-driven discrete simulator developed to study the scheduler Omega [Schwarzkopf et al., 2013], which we extended[10] in order to make it work with applications, instead of low-level jobs and to use the concept of component classes. Our scheduler implementation supports a variety of policies, from the basic FIFO (First In, First Out), to the size-based disciplines in the family of SMART policies [Wierman et al., 2005]. In case of size-based policies, we assume application size information to be provided by an external component (such as a job size estimator), which is not part of the design of our solution: recent studies have highlighted that a size based scheduler may tolerate estimation errors with minimal impact on the scheduling performance [Dell'Amico et al., 2016].

Our implementation first obtains a "virtual assignment" with Algorithm 1, then fulfills it by allocating resources accordingly, which happens instantaneously. Additionally, we have implemented a baseline, consisting of a rigid scheduler that does not distinguish component classes, which is representative of current cluster management systems. In our simulations, we consider two-dimensional resources, including definitions of CPU and RAM requirements. We would like to stress that the "virtual assignment" can take into consideration other constraints as well (e.g., GPU).

Our scheduler currently accepts **application workloads** of two kinds. The first is **batch applications**, that take from a few seconds to a few days to complete: these are delay-tolerant applications, with

[10] https://github.com/DistributedSystemsGroup/cluster-scheduler-simulator

a very simple life-cycle. Core components must first start to produce useful work, by executing user-defined jobs that are "passed" to the application; elastic components may contribute to the application progress. Once the user programs are concluded, the application finishes, releasing resources occupied by its frameworks and components. The second is **interactive applications**, which involve a "human in the loop": these are latency-sensitive applications, with a life-cycle triggered by human activity. In this case, core components must start as soon as possible, to allow user interaction with the application (e.g., a Notebook).

For our performance evaluation, we use publicly available traces[11] [Reiss et al., 2012, 2011; Wilkes, 2011], and generate a workload by sampling the empirical distributions we compute from such traces. First, we focus on batch applications alone, and simulate both *rigid* (e.g. TensorFlow) and *elastic* (e.g. Spark) variants: the label **B-R** represents rigid applications with only core components; the label **B-E** stands for elastic applications, with both core and elastic components. Then, we evaluate the benefit of preemption by complementing the above workload with (simulated) interactive applications.

Figure 3.2 shows the characteristics of the workload, and in particular the CDFs of the main metrics. The *application inter-arrival times* exhibit a bi-modal distribution with fast-paced bursts, as well as longer intervals between application submissions. The application *runtime* ranges from a few dozen seconds to several weeks (of simulated time). Looking more in details within jobs, it is possible to see that resource requirements take up to 6 cores (fig. 3.2 central left) and range from few MB to a few dozens GB of memory (fig. 3.2 central right). Finally, application components can be divided into *core components* and *elastic components*: batch applications consist of few to tens of components, up to thousands of components, while interactive applications are smaller, and use up to hundreds of elastic components.

The workload used in our simulations consists of 80,000 applications, with 80% batch and 20% interactive applications. Batch applications include 80% elastic and 20% rigid components. We simulate a cluster with 100 machines, each with 32 cores and 128GB of memory. All results shown here include 10 simulation runs, for a total of roughly 3 months of simulation time for each run.

Finally, the metrics we use to analyze the results include: application **turnaround** and **queuing time**, the latter being an important factor contributing to the turnaround time. Additionally, we measure the **queue sizes** and the number of running applications, along with the **resource allocation**, measured as the percentage of CPU and memory the scheduler allocates to each application.

3.3.2 Comparison with the baseline

We now perform a comparative analysis between our flexible scheduler and the baseline. In this series of experiments we omit interactive applications, and thus disable preemption. In order to show the benefits of our scheduler, we present results for a size-agnostic policy – FIFO – and for a size-aware policy – Shortest Job First (SJF).

[11] https://github.com/google/cluster-data

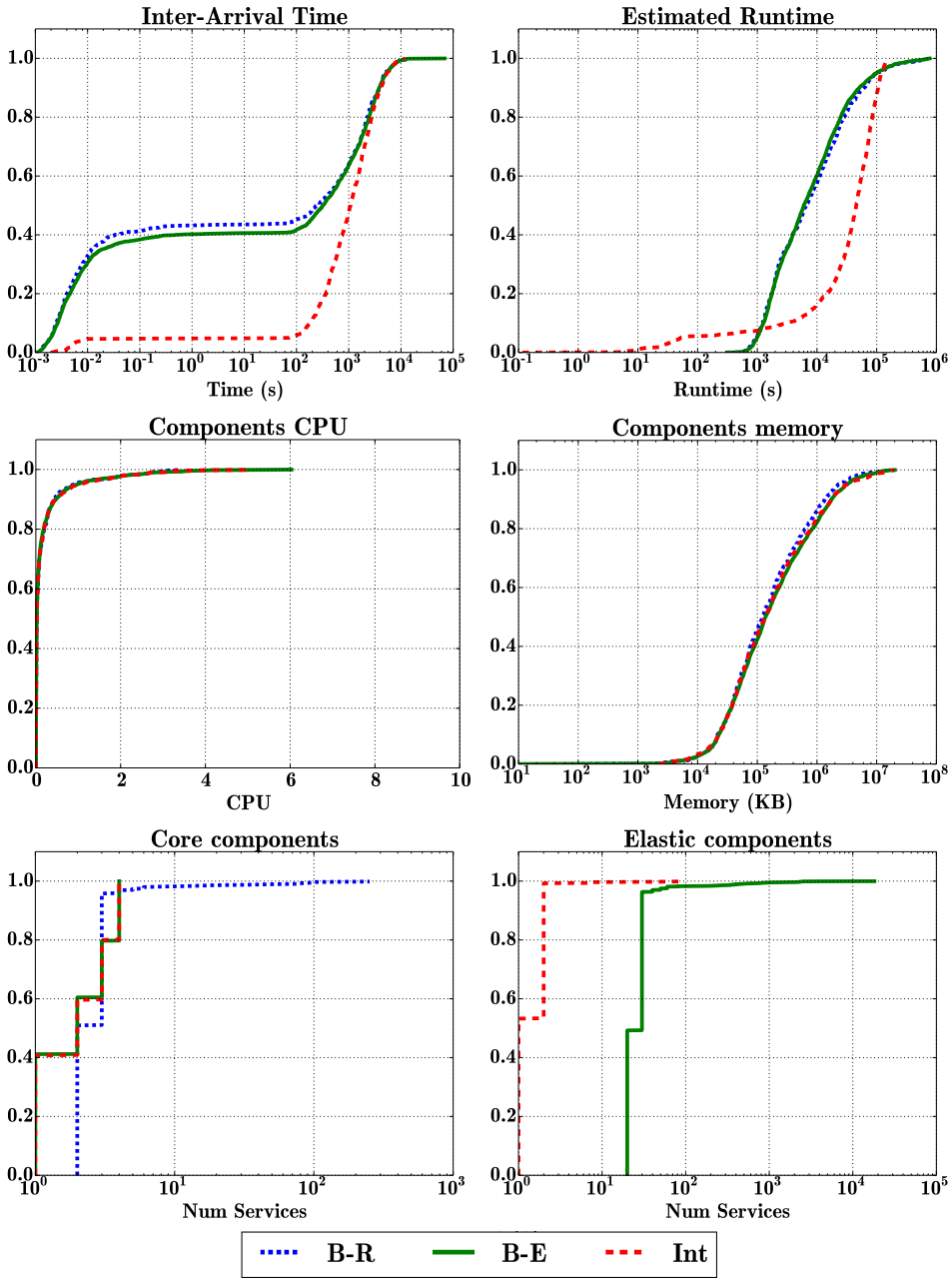


Fig. 3.2 Workload Definition: CDFs of different metrics.

Figure 3.3 (left) illustrates the most important percentiles (in a box-plot) of the distribution of turnaround times. The benefits of our approach are noticeable, irrespectively of the scheduling discipline: the median turnaround is halved when compared to the baseline, indicating superior system responsiveness. Additionally, we observe the benefits of a size-based policy in further decreasing turnaround times. We note that our approach is beneficial for both rigid and elastic batch applications: Figure 3.3 (center) shows a box-plot of application queuing times, which contribute to their turnaround. With our approach,

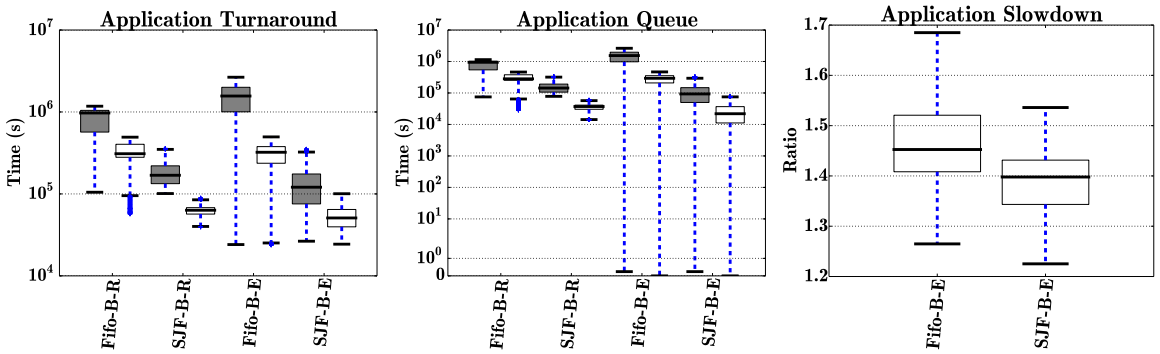


Fig. 3.3 Comparison of turnaround and queue time distributions, and application slowdown distributions for FIFO and SJF policies. White boxes (right box of every pair) corresponds to our flexible scheduler, gray boxes correspond to the baseline. *B-E* stands for batch elastic and *B-R* stands for batch rigid applications.

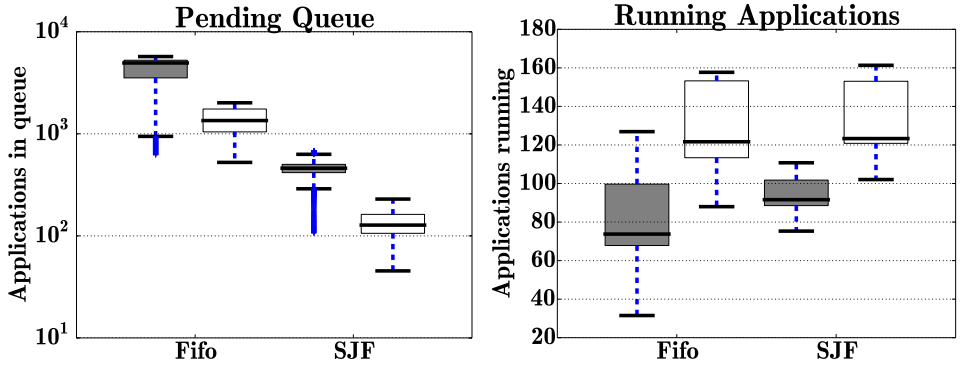


Fig. 3.4 Comparison of queues size for FIFO and SJF between our flexible scheduler and the baseline. The white boxes (right box of every group) correspond to our flexible algorithm, gray boxes to the baseline.

both kinds of applications spend less time waiting in a queue to be served. By differentiating classes of components, applications can execute as soon as enough resources to produce work are available. Finally, Figure 3.3 (right) focuses on application runtime: we report the **slowdown** computed as the ratio between the nominal application runtime (*i.e.*, the time required for an application to complete in an empty system, with all application components allocated their requested resources) and the effective application runtime obtained with the simulation. Values above one indicate that applications run slower in a system absorbing a given workload when compared to applications running in an empty system. Overall, these results show that our scheduling approach does not impose a high toll on application runtime, while globally contributing to improved turnaround times.

Next, we support the general results discussed above with additional details. Figure 3.4 shows the box-plots of the distribution of queue sizes, for both the pending and the running queues. Our approach induces a smaller number of applications waiting to be served, as well as a larger number of applications running in the system, compared to the baseline and across different policies. Indeed, our flexible

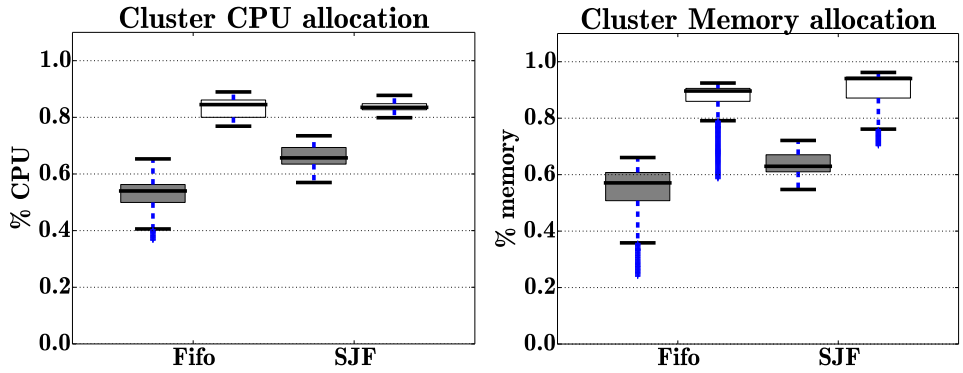


Fig. 3.5 Comparison of resource allocation distributions for FIFO and SJF policies, between our flexible scheduler and the baseline. White boxes (right box of every pair) correspond to our approach, dashed boxes to the baseline.

scheduler achieves a better packing of applications, which means they can start sooner. Additionally, the benefits of a size-based discipline are clear: the number of applications waiting is almost one order of magnitude smaller compared to a FIFO policy, while the number of running applications is similar.

Figure 3.5 shows metrics from the cluster perspective: our approach (for both disciplines) induces a far better resource allocation compared to the baseline, achieving more than 20% gains in both CPU and RAM allocation.[12]

3.3.3 Comparison with a malleable scheduler

The illustrative example depicted in Figure 3.1 shows that a rigid scheduler may not be able to exploit all the resources, therefore the results presented in the previous section are simple to understand and explain. One may wonder if a completely malleable scheduler may be able to actually use most of the resources, without requiring a more complicated scheduler. It is worth mentioning that currently no solution supports a malleable scheduler as we presented it in section 3.1.2. In other words, we have implemented a malleable scheduler in order to compare its performance with our flexible scheduler. The malleable scheduler uses a first-fit approach, adding as much elastic components as possible. If the remaining resources are not sufficient to schedule at least the rigid components of the next application, the scheduler waits for free resources before scheduling that application.

As we noted with the illustrative example (Figure 3.1), the average turnaround time with a malleable approach is similar to the one obtained by our flexible approach: the key advantage of our approach is in exploiting more efficiently the resources. This is confirmed by the experiments with our simulator.

In Table 3.1 we compare average turnaround and the average cluster allocation when using the malleable scheduler and our flexible scheduler, in case of FIFO and SJF. Our flexible scheduler is able to improve the turnaround time by 10% and 16% for FIFO and SJF respectively since it is using more

[12] Allocation is different from utilization: the simulator does not account for real application execution, so we cannot report utilization figures.

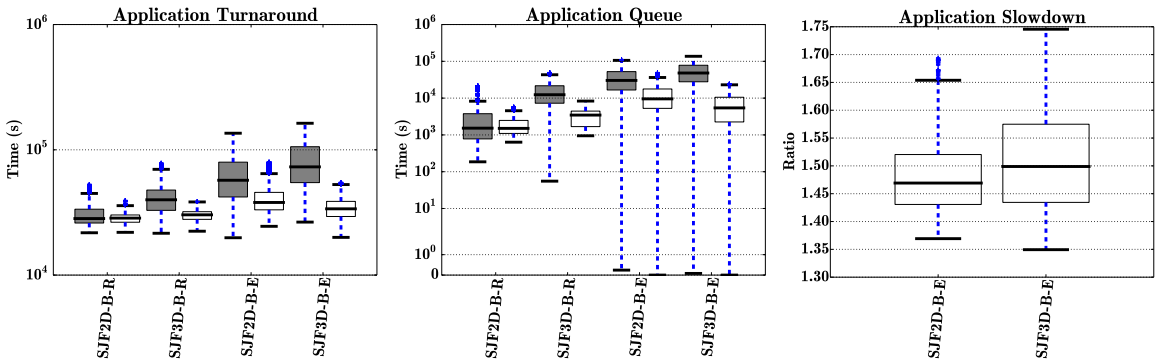


Fig. 3.6 Comparison of turnaround and queue time distributions, and application slowdown distributions for the SJF policy with different definitions of size. White boxes (right box of every pair) corresponds to our flexible scheduler, gray boxes correspond to the baseline. *B-E* stands for batch elastic and *B-R* stands for batch rigid applications.

Table 3.1 Comparison of average turnaround, CPU and Memory allocation for FIFO and SJF policies, between our flexible and a malleable scheduler.

Policy	Schedulers	Turnaround	CPU	Memory
FIFO	Malleable	$3.72x10^5$s	75%	74%
FIFO	Flexible	$3.36x10^5$s	77%	76%
SJF	Malleable	$6.14x10^4$s	74%	72%
SFJ	Flexible	$5.13x10^4$s	80%	78%

efficiently the available resources. This is confirmed by the CPU and memory allocation: our scheduler consistently uses more CPU and memory than the malleable scheduler.

3.3.4 Comparison between different definitions of size

When dealing with monolithic applications, the definition of *job size* is simple: it may be computed as the time necessary to complete the job when it runs in isolation, *i.e.*, without any interference caused by other jobs. This is actually the definition we have used in the previous section when using a size-based scheduler.

The above definition, nevertheless, may not capture the complexity of a distributed application. For instance, let us consider two applications with the same runtime, but different number of parallel tasks to perform: do they have the same size? Intuitively, the application with fewer tasks should be smaller than the application with a larger number of tasks.

This example suggests that it is possible to define the size of an application in different ways. In general, we may consider the *total amount of work* that the application needs to do. Therefore, similarly to [Schwiegelshohn and Yahyapour, 1998], a possible definition of size may be the product of the runtime *and* the number of components. Another option, if we consider all resources required by an

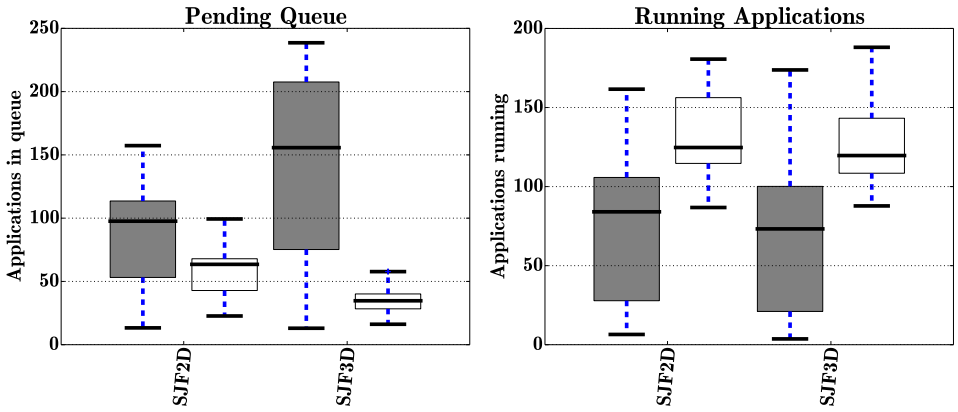


Fig. 3.7 Comparison of queues size for the SJF policy with different definitions of size. White boxes (right box of every pair) corresponds to our flexible scheduler, gray boxes correspond to the baseline.

Table 3.2 Definition of size used in the evaluation

Name	Definition
SJF	$runTime$
SJF-2D	$runTime * \#components$
SJF-3D	$runTime * \sum_{i=1}^{components} CPU_i * RAM_i$

application, is to compute the size as the product of the runtime, the number of CPUs and the memory of all the components. Table 3.2 summarizes the different definitions of job size for the SJF policy. Although in this section we focus on the SJF policy, we found that the same definitions can be applied to other policies as well.[13]. The definitions of job size that we take into consideration incrementally add information. We start by the classic definition that consider only the runtime of the application. Next, since that definition is one dimensional, we try to move to a two-dimensional definition by adding also the number of components that the application is using. Finally, since every component might request different resources, we add this information as well and turn the definition from a two to a three dimension.

Note that no definition is better than the other: it depends on the metric used to evaluate the system performance. If the focus is to minimize the turnaround time, then the basic definition of size (as runtime) may be sufficient. The following example shows why. Consider two applications, A_1 and A_2, whose requests arrive at the same moment. The core components of the two applications need $C_1 = 10$ units and $C_2 = 6$ units respectively, while they both have no elastic components. The run times are $T_1 = 2$ and $T_2 = 3$ seconds respectively. The system has 10 available resource units, therefore the two applications can not be scheduled in parallel. If we use the runtime as job size, we schedule application A_1 first then A_2, with an average turnaround time equal to 3.5 seconds. If we use as size the product of

[13] More information and experiments can be found in [Pace, Venzano, Carra and Michiardi, 2016]

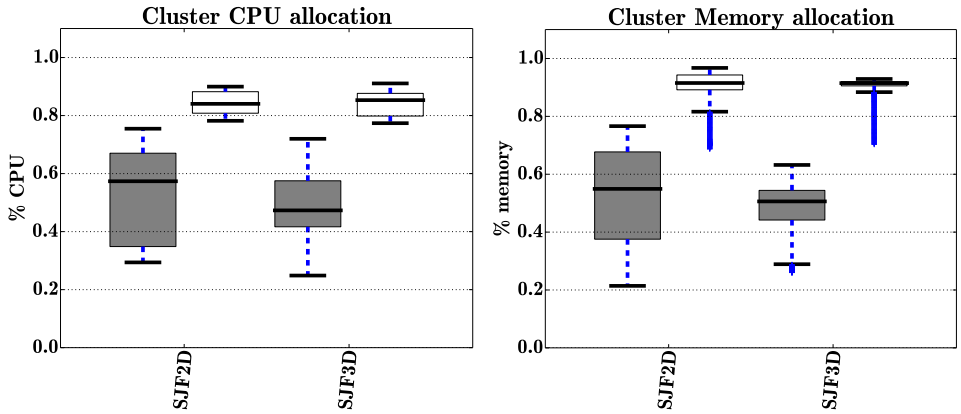


Fig. 3.8 Comparison of resource allocation distributions for the SJF policy with different definitions of size. White boxes (right box of every pair) corresponds to our flexible scheduler, gray boxes correspond to the baseline.

the runtime and the number of components, then we schedule the application A_2 and then A_1, with an average turnaround time equal to 4 seconds.

In the following, we compare the results obtained with different definition of size. We are not interested in the absolute values of the performance metrics: instead, given a definition of size, we compare the baseline scheduler with our flexible scheduler, in order to show the improvements we are able to obtain. The experimental settings are the same used in Section 3.3.2

Figure 3.6 shows the application turnaround, queuing time and slowdown. Focusing on the application turnaround, we notice that our flexible scheduler consistently improves over the baseline scheduler in case of batch elastic applications. For the batch rigid applications instead, while the "-3D" variant shows an improvement over the baseline scheduler, the two schedulers obtain similar results with the "-2D" variant. This is reflected on the application queue, which is similar for the two schedulers with the "-2D" variant. Nevertheless, the overall mix of the applications is able to exploit the resources more efficiently. In fact, as shown in Figure 3.7, right hand side, there are more application running, and, as shown in Figure 3.8 these applications use almost fully the CPU and the memory. In addition, by comparing the results from fig. 3.6 to fig. 3.3, we notice that different and more fine grained definitions of application size lead to an improvement in the average turnaround time when considering our flexible approach; average turnaround time of SJF-3D $<$ SJF-2D $<$ SJF. Instead, for the baseline approach this is not always true; the SJF-3D performs worst compared to SFJ-2D.

In conclusion, with these experiments we show that more information embedded in the definition of application size might translate in tangible benefits in terms of turnaround times, provided that the scheduling algorithm, like ours, can use such information. This is not always true when considering a baseline approach because the constraints imposed by it, like scheduling all components or nothing, void the benefits of a fine grained size definition.

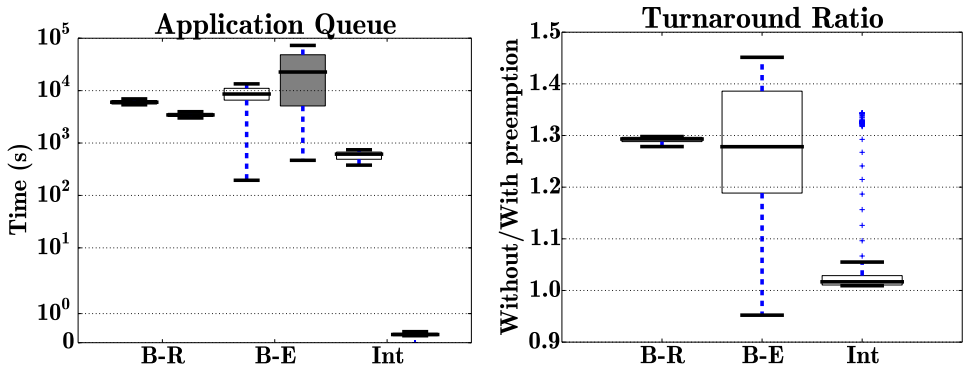


Fig. 3.9 On the left, comparison of queuing time distributions between scheduling with and without preemption. White boxes (left box of every pair) correspond to a *non-preemptive* system, gray boxes to our preemptive algorithm. On the right, turnaround ratio distributions between scheduling with and without preemption. *B-E* stands for batch elastic applications, *B-R* stands for batch rigid applications and *Int* is for interactive applications.

3.3.5 Preemption

We turn now our attention to the full workload we defined in Section 3.3.1, including *interactive* applications. Preemption is used when a high-priority, interactive application requires resources: this applies both to manually set priorities (e.g., in a FIFO policy) and to size-based priorities. In this section we focus on sized-based policies: in particular, we report results for the SRPT policy, which is a preemptive policy, and we use the runtime as definition of size.

Figure 3.9 shows the most relevant percentiles of the distribution of application queuing times, grouped by application type (both cases of batch and interactive applications), with and without our preemption mechanism. Globally, preemption does not subvert the perceived system responsiveness. However, interactive applications under preemptive scheduling enjoy roughly two orders of magnitude less queuing times. Users do not wait for few dozens minutes but only few seconds, for their interactive application to start. As a consequence, elastic batch applications pay with more variability (but stable for the median case) in queuing times.

Since our simulator does not account for real work done by applications, the preemption mechanism does not have any effect on the work that has been done by preempted components. In practice, our current preemption mechanism would instead suppress work done by elastic services, if preempted. Studying new preemption primitives, e.g. by suspending Linux containers, is part of our research agenda: this is the main reason why our current prototype implementation lacks support for preemption.

3.3.6 Additional considerations

In the previous sections we have shown the results for a size-agnostic policy, namely FIFO, and a size-aware policy, SFJ. Additionally, when presenting the results in case of preemption, we have considered the SPRT policy. SJF and SRPT are two examples of SMART policies [Wierman et al., 2005], which

are a set of policies whose aim is to minimize the average turnaround time. Another example of a smart policy is Higher-Response-Ration Next (HRRN): HRRN aims at avoiding the starvation of long running applications – starvation that may appear when using SJF or SRPT – by using the concept of virtual size.

We have tested our scheduler with all the policies mentioned above (SJF, SRPT, HRRN), in all the different scenarios: comparison with the rigid, as well as malleable schedule, when no interactive applications are present; comparison with the different definitions of size; comparison when preemption is enabled, comparison with different workloads. We do not report here all the set of results since they yield the same information of the results presented in the previous sections – the interested reader can find them in our Technical Report [Pace, Venzano, Carra and Michiardi, 2016]. In summary, our flexible scheduler is able to reduce the turnaround time, while improving resources allocation, in all the different policies.

3.4 Implementation: The Zoe system

Next, we describe Zoe[14], the system we have built to materialize the concepts developed earlier.

Zoe allows defining *analytics* applications and schedule them in a cluster of machines. It is designed to run on top of an existing low-level cluster management system, which is used as a back-end to provision resources to applications. Raising the level of abstraction to manipulate analytics application is beneficial for users and ultimately to the system design itself: application scheduling decisions can be taken with a small amount of state information, and do not happen at the same (extremely fast) pace as low-level task scheduling. Next we overview Zoe's design, and provide relevant details for the subject of this chapter.

Zoe applications In Zoe, the concepts introduced in Section 3.1 take the form of simple JSON description files that follow a high-level configuration language (CL) to specify applications, frameworks and components with their classes (core or elastic), resource reservations and constraints. The CL is simple and extensible: it aims at conciseness and, with framework templates, can be used by "casual" and "power" users [Verma et al., 2015].

The key aspect that determines the application type (batch, interactive, or any new type) is the way application life-cycle is managed. This is determined by a flexible attribute, reminiscent of a "command line", which allows passing runtime configuration options, user-defined arguments and environment variables, as well as setup and cleanup procedures. For application design, the "command line" attribute requires minimal knowledge of the frameworks that constitute an application. As an example of the simplicity and effectiveness of the Zoe CL, building a batch application for the distributed version of TensorFlow only requires tens of lines of CL. In this case, the most important attribute is the "command line", which is required to run a TensorFlow program, *i.e.,* `python $TF_PROGRAM`

[14]Zoe, https://zoe-analytics.eu/, was conceived in August 2015, named after the biggest container boat in the world, which, has touched sea (https://www.marinetraffic.com/en/ais/details/ships/209870000) in the same period of time. In this chapter, we omit several implementation details that stem from our continuous effort to extend Zoe.

`$PS_HOSTS $WK_HOSTS program-args`. Environment variables are appropriately handled by Zoe, including information unknown at scheduling time (e.g., host names).

A note on application failures is required. Any failure of an elastic component is practically harmless, whereas core component failures imply application failure. An area of future work is to exploit failure tolerance mechanisms available from some back-ends (e.g., Kubernetes) to steer application-level failure tolerance modes.

Zoe back-ends The main design idea of our system is to hide the complexities of low-level resource provisioning from application scheduling and exploit an existing cluster management system, for which many alternatives exists. Currently, Zoe builds on top of Docker Swarm, and uses it to achieve a series of objectives we list below:

- *Orchestration*: Zoe interacts with all the machines in a cluster using the Docker orchestration API (known as Swarm), which governs the behavior of the Docker engine deployed in each machine. Thus, Zoe manages to distribute the necessary binaries for the components of an application that is scheduled for execution, their configuration, life-cycle, and provisioning.
- *Dependency management*: Zoe applications materialize as a series of Docker images, which contain all dependencies and external libraries required for an application to run. Zoe applications can be built from *community-provided* or custom Docker images of existing frameworks.
- *Resource isolation*: framework components specified in an application run in Linux containers, which are managed by a Docker engine. We also use the Docker engine to achieve memory allocation, whereas CPU partitioning is left to the machine OS. This means, we have a one dimensional packing problem.
- *Resource matching*: application descriptions include resource constraints. When an application is scheduled for execution, Zoe instructs the back-end to adhere to component constraints when provisioning the relevant Docker images with framework binaries, as determined by the virtual assignment obtained by Algorithm 1.
- *Naming and networking*: the services for application components to cooperate in producing useful work, and to interact with the outside world are an important aspect to consider when choosing an appropriate back-end for Zoe. We use Docker networking, but we also have developed our own service discovery mechanism to allow a more flexible application configuration and deployment.

Zoe architecture Although Zoe is separated in several modules, it does not require any cluster-wide installation, because it uses its back-end to interact with the cluster.

The Zoe *master* polls a high-fidelity view of the cluster state through its back-end, whenever the scheduler is triggered, and stores it into a *state store*, backed by a PostgreSQL database. The state store also holds applications state, which is modeled as a simple state-machine. Because Zoe handles high-level objects (applications), the strain on the system is minimal: the rate of scheduling decisions scales

well even with heavy workloads. The virtual assignment procedure avoids *application interference* by construction because it considers requests in sequence, according to their priority. The virtual assignment is imposed on the back-end, using its API.

The Zoe *client API* handles REST calls that mutate the system state, or that can be used to monitor the system behavior. Command-line and web interfaces allow users and administrators to interact with the system and the cluster.

The Zoe *scheduler* implements the algorithm described in section 3.2. When an application is submitted, the Zoe master creates an entry in the application state store, and adds it to a *pending queue*. Our system allows plugging several *scheduling policies* to manage the pending queue, ranging from simple to sophisticated size-based policies. Such policies determine which application is granted "access" to cluster resources: to this end, the scheduler uses the cluster state store to simulate possible deployments before accepting an application. Framework components underlying an application are scheduled according to their type. The scheduler strives at making sure the application selected for execution can make progress as soon as resources are allocated to it. The Zoe *monitoring* module uses the Docker event stream to update the state of each application component running in the system.

Currently, the Zoe system supports a naive *preemption mechanism*: entire applications can be killed upon a command. The finer strategy described in Section 3.2 and Section 3.3 is currently under development.

Finally, although Zoe supports many data sources and sinks, we report experiments using a HDFS cluster to store input data to applications, and CEPH volumes to store application-specific logs.

3.5 Experiments with Zoe

Our goal now is to perform a comparative analysis of two generations of Zoe: the first, implementing a rigid scheduler, as for the baseline, the second with the flexible scheduler we present here. In our experiments, we replay the exact same workload trace for both generations. Each trace takes about 3 hours from the first submission to the last. During our experiments, no other user was allowed to submit jobs to Zoe.

Workload We use two representative *batch application templates*, including: 1) an elastic application using the Spark framework; 2) a rigid application using the TensorFlow framework. Following the statistical distribution of our historical traces, we set our workload to include 80% of elastic and 20% of rigid applications, for a total of 100 applications. Application inter-arrival times follow a Gaussian distribution with parameters $\mu = 60$ sec, and $\sigma = 40$ sec, which is compatible with our historical data. More specifically, using the elastic application templates, we run two use cases. First, an application to induce a random-forest regression model to predict flight delays, using publicly available data from the US DoT.[15] Second, a music recommender system based on alternating least squares algorithm, using

[15] http://stat-computing.org/dataexpo/2009/the-data.html

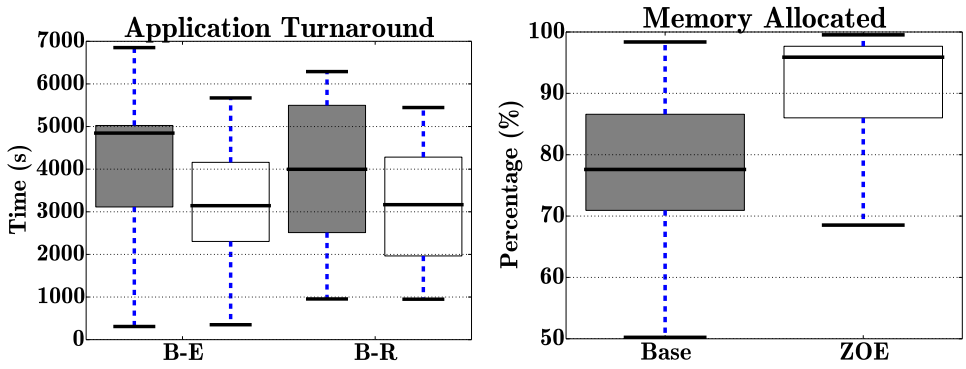


Fig. 3.10 Comparison of turnaround time distributions using the FIFO discipline. White boxes (right box of every pair) correspond to the second generation of Zoe that implements our algorithm. *B-E* stands for batch elastic and *B-R* stands for batch rigid applications.

publicly available data from Last.fm[16]. Both applications have two different requirements (flavors) in term of memory for each elastic component. The random-forest regression model has 3 core components and 32 elastic components of 16GB or 8GB RAM each (depending on the flavor); every elastic component uses 1 CPU. The music recommender system has 3 core components and then 24 elastic components of 16GB RAM or 8GB each (depending on the flavor); every elastic component uses 6 CPU. Instead, using the rigid application template, we train a deep Gaussian Process model [Cutajar et al., 2016], and use both a single-node and a distributed TensorFlow program, requiring 1 and 10 workers (and 5 parameter servers) each with 16GB of RAM.

Experimental setup We run our experiment on a platform with ten servers, each with two 16-core Intel E5-2630 CPU running at 2.40GHz (total of 32 cores with hyper-threading enabled), 128GB of memory, 1Gbps Ethernet network fabric and ten 1TB hard drives. No GPU-enabled machines are available in our platform, at the moment. The servers use Ubuntu 14.04, Docker 1.11 and the standalone Swarm manager. Docker images for the applications are preloaded on each machine to prevent container startup delays and network congestion.

Summary of results Using the FIFO scheduling policy, we compare the two generations of Zoe according to the distributions of application turnaround times, as shown in Figure 3.10 (left). The behavior of the two systems indicate a clear advantage for our approach: the median turnaround times are 37% and 22% lower, for elastic and rigid applications respectively. Note also that the tails of the distributions are in favor of our approach.

Overall, the new generation of Zoe that implements the flexible scheduler is more efficient, with a 20% improvement, in allocating and packing applications, as illustrated in Figure 3.10 (right), where we show the ratio of the distribution of allocated over available resources.

[16] http://www-etud.iro.umontreal.ca/~bergstrj/audioscrobbler_data.html

Finally, we present results concerning a low-level metric that measures the application *ramp-up time*, *i.e.,* the time it takes for applications scheduled for running, to receive their allocations and start producing work. Zoe achieves a container startup time, including placement decisions, of $0.90s \pm 0.25ms$. Full-fledged applications, made by several containers, only take few seconds to start, which is a compelling property, especially when compared to existing solutions such as Amazon EMR.

3.6 Summary

Efficient resource management of computer clusters has been a long-lasting area of research, with peaks of attention happening in conjunction to improvements in computing machinery, e.g. lately with cloud computing and big data. A new breed of cluster management systems, aiming at becoming "data-center operating systems", are currently been confronted with problems of efficiency and performance at scale.

Despite recent advances, there exists a gap between the goal of low-level resource management, and that of manipulating high-level, heterogeneous, distributed (analytic) applications running in such cluster environments. In this chapter we presented a first possible step to fill this gap, in the form of a new application scheduler that interacts with a cluster management back-end, to schedule and allocate resources to applications defined with a simple language and semantics. In addition to careful engineering, required to design and implement our system we call Zoe, our research identified a more fundamental problem, that calls for a novel scheduling heuristic capable of manipulating composite applications, while contributing to system responsiveness.

We validated our algorithm to address our scheduling problem along two lines. We used a numerical approach to simulate large-scale deployments and workloads. We showed our scheduling algorithm to be highly effective in reducing turnaround times, in particular by reducing applications queuing times. Consequently, cluster resources were better allocated. In addition, we reported an overview of the evaluation of Zoe, that indicates superior performance and efficiency related to our flexible scheduling heuristic.

Next we are going to focus on the development of a method to redeem untapped resources from idle but running applications.

Chapter 4

Data-Driven Resource Allocation

Despite recent efforts in the research area of resource allocation and scheduling, data-center resources go often under utilized, as shown in recent traces from large-scale production clusters [Reiss et al., 2012; Wilkes, 2011]: Figure 1.1 illustrates resource utilization in a operational cluster at Google, for a mixed workload of production services and batch applications; in most cases ($\sim$ 80%) resource utilization is less than 40% or 80% of the allocated resources depending on different application types[1].

Current approaches that address efficiency requirements fall in two broad categories. The first involves methodologies that steer tenants' behavior through the design of incentive mechanisms; tenants are endowed with the task of optimizing their cost to operate their applications, whereas providers operate on prices to regulate the allocation of idle resources. Such approaches are largely adopted by public cloud providers [Babaioff et al., 2017]. The second category concerns approaches that operate at the system level, and propose mechanisms that allocate resources based on tenants' reservations[2][3] [Ghodsi et al., 2011; Hindman et al., 2011; Rasley et al., 2016; Schwarzkopf et al., 2013; Verma et al., 2015]. The ultimate goal of this line of research is to render the concept of resource reservation obsolete, and either let tenants reason in terms of value and cost [Babaioff et al., 2017], or let the system determine how to avoid wasting precious and costly resources, especially when the latter are scarce and entail application queuing in the scheduler.

In this chapter, we discuss a methodology that essentially lies in the second category. We present a system that dynamically allocates resources according to historical observations of its utilization. More specifically, we leverage machine learning algorithms in order to adjust allocation to the expected utilization. We present our design of a data-driven scheduling mechanism that improves cluster utilization, thus decreasing the average turnaround time, while preventing application failures due to resource contention. Our approach monitors resource utilization and relies on sophisticated online resource demand forecasting to modulate allocated resources such as they approximate utilization patterns well.

[1] In the analysis we saw that some applications were using more resources than requested and this was confirmed by Google staff. Their system allows the user to go above the reservation when resources are available. Since not all the systems can do this (e.g.; Docker), we decided to remove that portion of the data.

[2] http://www.docker.com/

[3] https://aws.amazon.com/emr/

Our experiments, that we conduct on a system simulator as well as a prototype implementation using real-life data-center traces, indicate substantial gains over existing alternatives: our approach contributes to more efficient and responsive clusters, while carefully controlling the number of application failures due to the approximate nature of our control approach.

The remainder of this chapter is organized as follows. In section 4.2 we present our system design, and we validate our ideas using a simulation campaign in section 4.3. Finally, we present our prototype implementation in section 4.4 and its evaluation in section 4.5.

4.1 Problem Statement

In this thesis, we study the problem of cluster efficiency by reducing the resource slack induced by reservation-centric application schedulers, which match allocation to reservation. To do so, we introduce a new mechanism that **predicts** the resource utilization and adjusts the resource allocation accordingly. The main challenge to face is that prediction errors may have problematic consequences, since sudden spikes could wreak havoc the system [Verma et al., 2015]. When dealing with finite resources such as RAM, in fact, not providing the correct amount of resources leads to application failures. Careful engineering would suggest to introduce a buffer that will act as "safe-guard" to prediction errors. This results in a **trade-off**, since on the one hand the safe-guard buffer should be small to minimize slack, while on the other hand it should be sufficiently large to prevent application failures.

Previous works (a detailed description is provided in Section 2.1.3) usually consider shareable resources, such as CPU, where the effect of wrong resource dimensioning does not translate into application failures. Other approaches consider resource over-provisioning, where the slack is not continuously optimized, and where the application failures can be unpredictable and are taken care by the Operating System (OS).

In our approach, we leverage on three key ideas: prediction confidence, application elasticity and controlled failures. In the prediction process, most of the tools provide additional information about the **confidence** of the prediction. We use such information to dynamically adapt the safe-guard buffer that should prevent application failures. In addition, the frameworks, on which the applications are based, are composed by several elements that are characterized by either a **core** or **elastic** nature [Pace, Venzano, Carra and Michiardi, 2016]. Core components are compulsory for a framework to produce useful work (e.g, Apache Spark requires a controller, a master, and one worker); elastic components, instead, optionally contribute to a job, e.g. by decreasing its runtime. An application that features only core components is called **rigid**, whereas applications with a mix of core and elastic components are called **elastic**. If the resource demand is higher than the available resources, we intervene (when possible) on elastic components to avoid application failures. As a last step, should the previous two mechanisms not be sufficient to provide enough resources, we explicitly decide which application should fail so that to minimize the amount of wasted work.

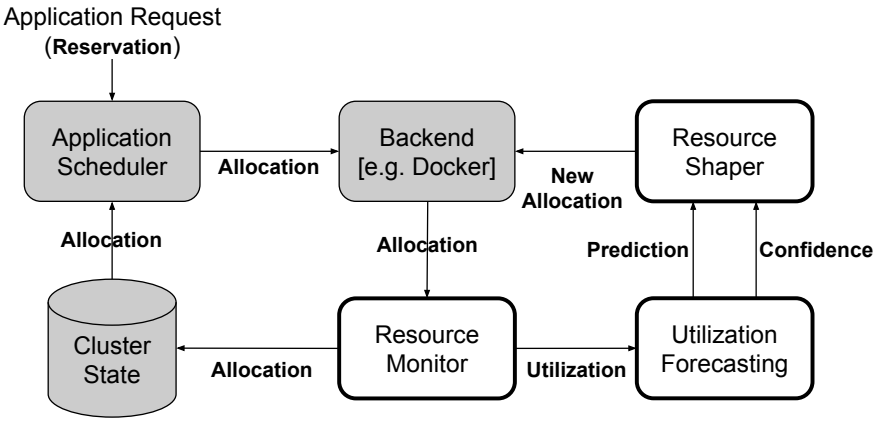


Fig. 4.1 System overview: shaded boxes represent existing components, white boxes indicate new components presented in this work.

4.2 System Design

The goal of our system is to increase cluster utilization and reduce average application turnaround times by adjusting allocation to track resource utilization by anticipating its dynamics, while reducing the number of "self-inflicted" application failures due to approximation errors.

Figure 4.1 illustrates the architecture we assume in our work. The *backend* module is an instance of a cluster management system, such as Docker[4] or Kubernetes[5], or alternative schedulers [Thinakaran et al., 2017]. Additionally, we assume the presence of an *application scheduler* such as [Pace, Venzano, Carra and Michiardi, 2016], which reads the compute cluster state from a dedicated database component. Finally, the monitoring component populates the cluster state database with measurements taken from the backend. In this Section, we focus on the two additional components we present in this paper: the *utilization forecasting* module, and the *resource shaper* module.

A bird's view on the operation of our system is as follows. Application execution requests take the form of resource reservations, which are submitted to the application scheduler. The application scheduler admits the request based on reservation information alone, and instructs the back-end to provision the necessary resources. The resource monitor collects information about both allocated and used resources, which are fed to the system state and the forecasting component respectively. The resource shaper module gauges resource allocation to match predicted utilization patterns, and is responsible for the preemption of running applications in case of sudden peaks in resource demand. The modified resource allocation is reflected in the system state, which in turn triggers new scheduling decisions. Next, we describe in detail the components that materialize our ideas.

[4] https://docs.docker.com/swarm/
[5] http://kubernetes.io/

Resource monitor This module collects information about resource allocation and utilization from *every* component of *every* running application. This happens at regular time intervals: higher frequencies provide more accurate views, but generate more data. Our goal is to minimize intrusiveness by being application agnostic: for this reason we do not instrument applications (as done for example in [Kuzmanovska et al., 2016]), but take standard metrics (CPU, memory, etc) as they are seen by the OS.

Utilization forecasting The goal of this module is to anticipate the resource utilization of every application component. We study both parametric and non-parametric modeling approaches to predict resource utilization, with emphasis on the quantification of the uncertainty associated to these predictions. A more detailed exposition of the methodology we employ can be found in Section 4.2.1.

Resource shaper This module uses utilization forecasts to adjust the resources allocated to every component of running applications. We anticipate prediction errors, thus we compensate using a "safe-guard" buffer of size β to artificially increase (that is, to force over estimation) predicted peak resource utilization. A more detailed exposition of β can be found in Section 4.2.1.

Additionally, the resource shaper is in charge of application preemption. Preemption policies can either be optimistic [Schwarzkopf et al., 2013; Verma et al., 2015] or strict (pessimistic). We advocate for a strict policy, to avoid delegating application preemption to the OS, which manages resource shortage (such as Out Of Memory (OOM)) in an application agnostic and "unpredictable" way. A detailed exposition of the preemption policy can be found in Section 4.2.2.

4.2.1 Utilization Forecasting Module

The forecasting module is responsible for making predictions about future resource utilization, for each application component. For a given application, we forecast *both* CPU and memory utilization using monitoring data, which is available in the form of a time series that reflects resource usage across time[6]. We seek to discover patterns of resource usage that allow reasoning about our expectations on the future state of the system utilization.

We advocate for the need to **quantify the level of uncertainty** associated with each prediction: predictive errors may have serious impact on "finite" resources (i.e. memory), as they can cause application failures. Although errors are unavoidable to a certain extent, predictive confidence can be used to adjust the degree of adaptiveness to the anticipated workload: intuitively, a prediction with low confidence implies that the resource shaper should be conservative regarding changes in resource allocation.

In this work we compare the traditional *parametric* Autoregressive Integrated Moving Average (ARIMA) model to an alternative *non-parametric* model that offers a principled quantification of uncertainty, which we introduced in Section 2.2. On the one hand, we use state-of-the-art ARIMA implementations that automatically tune hyper parameters and that provide a method to compute

[6]Other types of resource can be considered as well.

confidence levels associated to predicted values [Box et al., 2008]. On the other hand, we model resource utilization using Gaussian Process (GP) regression [Rasmussen and Williams, 2006], which is a Bayesian non-parametric regression method with many attractive features. Bayesian approaches control model complexity and thus avoid problems such as over-fitting [MacKay, 2003]. Moreover, GPs offer a sensible framework for tuning their hyper parameters, through evidence maximization, that does not require cross-validation approaches which are typically more expensive and unpractical in the context of our work. Finally, the output of a GP regression model is a predictive distribution, rather than a single prediction, which allows reasoning about uncertainty in a principled way.

How Online Forecasting Works

From a practical perspective, the *forecast component operates in an online manner*. As long as new data is available, the predictive model will be trained and subsequently queried about the future workload. Depending on the modeling methodology, our approach is as follows.

Using the ARIMA model The online training and prediction process that uses ARIMA operates by appending the new resource utilization data to the collection of observations gathered so far. ARIMA hyper-parameters are optimized using well-known methods[7][8], which are known to be computationally expensive. Alternatively, works like [Hyndman et al., 2007] propose a stepwise algorithm (instead of using grid-search) that improves performance.

The k-step ahead forecast error is a linear combination of the future errors entering the system after time t:

$$e_t(k) = y_{t+k} - \hat{y}_t(k)$$

where $\hat{y}_t(k)$ is the estimated value. Since $\mathrm{E}[e_t(k)|y_t] = 0$, the forecast $\hat{y}_t(k)$ is unbiased with Mean Squared Error (MSE):

$$\mathrm{MSE}[y_t(k)] = \mathrm{Var}[e_t(k)]$$

Given these results, if the process is normal, the $100(1-\alpha)$ forecast interval is:

$$[y_t(k) \pm N_{\alpha/2}\sqrt{\mathrm{Var}[e_t(k)]}]$$

where $N_{\alpha/2}$ is the multiplicative factor to obtain the percentile.

Using the GP model The online training and prediction process that uses GP regression operates as follows:

1. New resource utilization data is appended to the collection of observations $\mathbf{X}, \mathbf{y}$. The rows of $\mathbf{X}$ are patterns as defined in Equation (2.5).

[7] http://pyramid-arima.readthedocs.io/en/latest/_submodules/arima.html
[8] https://www.rdocumentation.org/packages/forecast/versions/8.3/topics/auto.arima

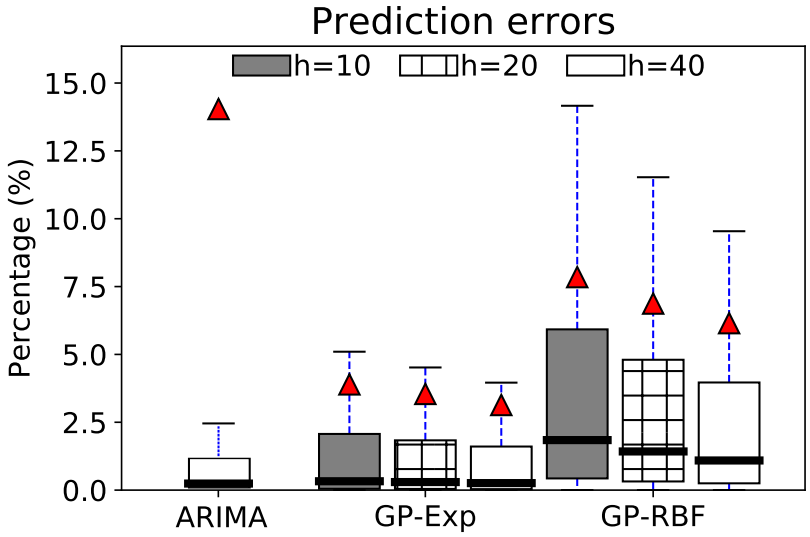


Fig. 4.2 Boxplot showing error distribution of predicted utilization for a collection of time series in our academic cluster with different history points and, in case of GP, different kernels. The red triangle is the mean.

2. Using a history-dependent kernel $k_h(x,x')$, Equations (4.1) and (4.2) are used to make predictions based on observations $\mathbf{X},\mathbf{y}$.

Under the assumption of a zero-mean prior and a Gaussian *likelihood*, that is, for any input-output pair we have $y \sim N(f(x),\sigma^2)$, the posterior is also a GP whose mean and covariance can be calculated analytically as follows:

$$\mathrm{E}[f(x) \mid \mathbf{X}] = k_h(x,\mathbf{X})(k_h(\mathbf{X},\mathbf{X}) + \sigma^2)^{-1}\mathbf{y} \tag{4.1}$$

$$\begin{aligned}\mathrm{Var}[f(x) \mid \mathbf{X}] = {} & k_h(x,x') \\ & - k_h(x,\mathbf{X})(k_h(\mathbf{X},\mathbf{X}) + \sigma^2)^{-1}k_h(\mathbf{X},x)\end{aligned} \tag{4.2}$$

The predicted value at a new point will be the expectation under the posterior distribution, and the posterior variance quantifies the uncertainty about the prediction.

The regression step can be computationally expensive. Equations (4.1) and (4.2) involve a matrix inversion (for $k(\mathbf{X},\mathbf{X})+\sigma^2$), which is an operation of cubic complexity. Moreover, the set of observations $\mathbf{X},\mathbf{y}$ will grow indefinitely during the lifetime of the system. While there is a plethora of methodologies on sparse GPs in the literature [Chalupka et al., 2013; Quiñonero Candela and Rasmussen, 2005; Rahimi and Recht, 2007; Snelson and Ghahramani, 2005], that can be used to reduce the complexity of regression, in this work we adopt the simple solution of restricting the dataset $\mathbf{X},\mathbf{y}$ to the N latest observations, thus keeping the model tractable. Note that N is the number of patterns used; it should not be confused with h, which is the size of each pattern.

Numerical results We have applied our modeling approaches on a dataset consisting of approximately 6000 time series that monitor the memory usage of applications in our academic cluster. Figure 4.2 summarizes the empirical distribution function for the predictive errors observed across the entire dataset, using ARIMA and GP.

In case of GP we forecast the future value using different number of past observations $h = [10, 20, 40]$, with $N = h$. As seen in Figure 4.2, increasing the value of h results in smaller prediction errors. Also for the implementation of the history-dependent kernel as described in Equation (2.6), we have experimented both with the exponential and the squared-exponential (also known as RBF in the literature) functions. Figure 4.2 implies that the exponential implementation (GP-Exp) outperforms the RBF (GP-RBF) choice in terms of prediction error. Results for the GP are in line with our expectations, as the time series in question are typically not smooth. For the experiments of Section 4.3 and Section 4.5, we consider the exponential implementation of the history-dependent kernel only.

With ARIMA we observe that setting $p = h$ (so the autoregressive order equal to the history size) is overridden by hyper-parameter optimization, which yields $p \leq 3$. Hence, the results for ARIMA do not depend on h. From Figure 4.2, it appears that ARIMA performs slightly better compared to GP for the median test error. Also the variance of the predictive error is smaller than with the GP model, an indication of a possible "over-confidence" in the model predictions. Our experimental results discussed in Section Section 4.3 corroborate this intuition: over-confidence leads to higher application failure rates, and an overall lower system efficiency, when compared to the GP model we present in this work.

4.2.2 Resource Shaper Module

We now delve into the details of the resource shaper module, which we use to adjust resource allocated to an application and its components as a function of predicted utilization. When resource are underutilized, the resource shaper "redeems" the excess capacity such that the application scheduler can dequeue idle applications. On the contrary, upon a utilization spike, the resource shaper needs to redeem resources from running applications and dedicate them to those experiencing a peak demand, for otherwise such applications are doomed to fail. Thus, the goal of the *preemption policy* we associate to the resource shaper is to decide how to redistribute resources, by operating on running applications and their components. Such a policy can optionally account for application priorities, as dictated by the application scheduler. Note that, irrespectively of the chosen preemption policy, *a failed application is resubmitted to the application scheduler*, making sure it enters the scheduling queue in a position commensurate to its original priority.

Recent works (for example [Verma et al., 2015]) advocate for an *optimistic* preemption policy, which is reminiscent of optimistic concurrency control [Schwarzkopf et al., 2013]: resources are redeemed without taking explicit actions to manage the consequences of resource redistribution. Either explicit (and often manually set) priorities determine the fate of running applications, or the task is left to the OS.

Algorithm 3: Overview of the pessimistic preemption policy implemented by the resource shaper module.

Data: $\mathscr{H} \leftarrow$ Hosts, $\mathscr{A} \leftarrow$ Running Applications

$cpusFree \leftarrow Array(\mathscr{H})$
$memFree \leftarrow Array(\mathscr{H})$
foreach $host \in \mathscr{H}$ **do**
    $cpusFree[host] \leftarrow host.totalCpus$
    $memFree[host] \leftarrow host.totalMem$
$\mathscr{J} \leftarrow$ SORT($schedulingPolicy$, $\mathscr{A}$)
foreach $req \in \mathscr{J}$ **do**
    $cpus \leftarrow cpusFree$
    $mem \leftarrow memFree$
    $remove \leftarrow False$
    foreach $c \in req.CoreCpts$ **do**
        $cpus[c.host] \leftarrow cpus[c.host] - c.futureCpus - \beta$
        if $cpus[c.host] < 0$ **then**
            $remove \leftarrow True$
            break
        $mem[c.host] \leftarrow mem[c.host] - c.futureMem - \beta$
        if $mem[c.host] < 0$ **then**
            $remove \leftarrow True$
            break
    if $remove$ **then**
        INSERT(req, $\mathscr{K}$)
    else
        $cpusFree \leftarrow cpus$
        $memFree \leftarrow mem$
        $E \leftarrow$ SORT($timeAlive$, $req.ElasticCpts$)
        foreach $e \in E$ **do**
            $cpus \leftarrow cpusFree[e.host] - e.futureCpus - \beta$
            $mem \leftarrow memFree[e.host] - e.futureMem - \beta$
            if $cpus \leq 0$ ***or*** $mem \leq 0$ **then**
                INSERT(e, $\mathscr{K}_{\mathscr{E}}$)
            else
                $cpusFree[r.host] \leftarrow cpus$
                $memFree[r.host] \leftarrow mem$
foreach $req \in \mathscr{K}$ **do**
    foreach $c \in (req.CoreCpts \cup req.ElasticCpts)$ **do**
        PREEMPCOMPONENT(c)
foreach $e \in \mathscr{K}_{\mathscr{E}}$ **do**
    PREEMPCOMPONENT(e)
foreach $req \in \mathscr{J} \setminus \mathscr{K}$ **do**
    foreach $c \in (req.CoreCpts \cup req.ElasticCpts)$ **do**
        RESIZECOMPONENT(c)

Here, we present an alternative preemption policy, which we call *pessimistic*. Our goal is to control which application should be partially or fully preempted[9], while minimizing the amount of work that is wasted.

Algorithm 3 presents the details of our pessimistic preemption policy implemented by the resource shaper, which is triggered at regular time intervals, as determined by the output produced by the forecasting module. Given the current cluster state, and the resource utilization forecasts, the algorithm computes a new resource allocation for each running application, which is then imposed on the cluster by operating directly on application components through low-level preemption primitives.

The algorithm starts by initializing (lines 1-5) the variables that holds the information about the allocated resources. Then it sorts (line 6) running applications according to the application scheduler policy (e.g.; First-In-First-Out (FIFO), that is, arrival times), and it computes (lines 7-33) an allocation by trying to maximize the resource allocation while minimizing the number of running applications. In particular, it first allocates the core (lines 8-19) components and then all elastic components[10] that fit in the host (lines 23-33). The algorithm continues until all running applications are processed.

Resource allocation is determined, and we can turn our attention to preemption. Core components that no longer fit a host entail full application preemption (lines 34-36). Also elastic components can be preempted (lines 37-38), inducing only a partial application preemption. In addition, in case of elastics components, we can experience partial or entire loss of the work done by the preempted component. For this reason, our algorithm allocates the core components of an application, then moves to the elastic components by giving priority to the ones that have been living in the cluster for a longer time (line 25). Components recently scheduled are the best candidates for preemption, because they have likely produced less useful work. Finally, the algorithm resizes (lines 39-41) the components according to the computed allocations. Our algorithm currently supports CPU and Memory, but it can be extended to other types of resource as well.

Safe-guard buffer We are now ready to define the "safe-guard" buffer. The buffer size β is a function of the uncertainty quantified by the forecasting module:

$$\beta = K_1 R_{A_i} + K_2 V_{A_i} \tag{4.3}$$

where R_{A_i} is the initial resource **request** for application A_i, and V_{A_i} is the estimated variance of the prediction, as these are given by the forecasting module (ARIMA or GP). Equation (4.3) involves a constant term $K_1 R_{A_i}$ and a dynamic term $K_2 V_{A_i}$. The constant term can be though of as a *minimum resource allocation* that is granted to application A_i. The dynamic term uses the confidence (expressed as variance V_{A_i}) given by the predictor to adjust β accordingly: it thus changes during an application

[9] We consider preemption primitives such as a `kill` operation, which inevitably waste work. Component or application suspension [Pastorelli et al., 2014] and migration are outside the scope of this work. Alternatively, it would be interesting to consider techniques such as [Gu et al., 2017], which would allow a graceful management of memory pressure.

[10] In case the application scheduler does not support the distinction between core and elastic, all components are treated as core.

lifetime. In Section 4.3, we study how different values of K_1 and K_2 affect the performance of our method.

4.2.3 System Scalability

Scalability is a crucial aspect of systems that manage computer clusters. We comment on two sources of scalability bottlenecks in our solution: the algorithmic complexity *(i)* of the forecast module and *(ii)* of the algorithm to address resource conflicts.

Utilization forecasting Module The regression step of the predictor can be computationally expensive; the matrix inversion in eqs. (4.1) and (4.2) has cubic complexity on the set of observations recorded. Although we have already reduced the complexity by considering the N most recent patterns, it is still necessary to keep as many predictive models as the running components in the system. However, the execution of predictive models can be trivially distributed on every host of the cluster, since the models are independent and no coordination is required.

Resource Shaper Module From algorithm 3, we can see that its runtime complexity is $O(A \times C)$ where A and C are the cardinality of the application set and their components, respectively. Instead of having a single instance of the algorithm running in a central component, it is tempting to distribute the algorithm on a per-host basis, which contributes to reduced input sizes to the problem. However, a distributed approach lacks a global view on the system, and is thus prone to be sub-optimal when compared to a centralized algorithm. Local decisions can have global effects, e.g., preemption of a core component on a certain host entails preemptions of all related core components on other hosts. For that reason, we focus on the centralized variant. A competitive analysis of the distributed algorithm is subject of future research.

4.3 Numerical Evaluation

4.3.1 Methodology

We evaluate our algorithm using an event-based, trace-driven discrete simulator which was developed to study the scheduler Omega [Schwarzkopf et al., 2013], and was later extended in [Pace, Venzano, Carra and Michiardi, 2016] to study application schedulers. We have made additional extensions[11] to support the concepts of this work.

In this work, we are mainly interested in memory resources, which are much harder to manipulate than computational (CPU) ones. We use publicly available traces [12] [Reiss et al., 2012, 2011; Wilkes, 2011], and generate a workload by sampling from the empirical distributions computed from such traces. Our workload is composed by 150.000 batch applications, both *rigid* (e.g. TensorFlow) and *elastic* (e.g.

[11] https://github.com/DistributedSystemsGroup/cluster-scheduler-simulator
[12] https://github.com/google/cluster-data

Apache Spark) variants. Applications are assigned a number of components ranging from a few to tens of thousands. The resource requirements (in terms of memory) of application components follow that of the input traces, ranging from a few MB of memory to a few dozens of GB, and up to 6 CPU cores. Application *runtime* is generated according to the input traces, and ranges from a few dozens of seconds to several weeks (of simulated time). Inter-arrival times are drawn from the empirical distributions of the input traces, and exhibit a bi-modal distribution with fast-paced bursts, and longer intervals between application submissions.

We simulate a cluster consisting of 250 homogeneous machines, each with 32 cores and 128GB of memory. All results shown here include 10 simulation runs, for a total of roughly 3 months of simulation time for each run.

The metrics we use to analyze the results include: *(i)* application **turnaround**, which allows reasoning about the scheduling objective function, *(ii)* **resource slack**, measured as the difference of percentage of CPU and memory the scheduler allocates to each application compared to the percentage actually used by the application and *(iii)* application **failures**, which give us information about the aggressiveness of our approach. Note that we omit results that use size-based scheduling disciplines instead of simple FIFO. Clearly, both turnaround and slack metrics would improve, due to well known properties of size-based policies. The use of more sophisticated policies does not qualitatively change our observations on the merits of our dynamic allocation approach.

4.3.2 Results

Next, we present experimental results that demonstrate the advantage of our resource shaping mechanism, compared to a baseline approach which matches allocation to reservation. Two alternatives for time series prediction are examined. We first consider an ideal setup with an oracle having perfect information about future workload: this allows to determine an upper bound of the performance gains achieved by our approach. Then, we compare the two models developed in Sections 2.2 and 4.2.1 (ARIMA and GP), to investigate the impact of prediction errors on system performance.

Baseline It constitutes a reservation centric approach (similar to Mesos and Yarn, as originally implemented in the Omega simulator [Schwarzkopf et al., 2013]) that achieves the performance reported in Figure 4.3. This approach relies entirely on the resource requested by the application (when submitted) in order to allocate resources in the cluster and does not modify them at runtime. We show the difference in resource allocation compared to utilization, called slack, of CPU and memory. This result reiterates on the low efficiency of reservation centric approaches: the median resource slack is large (40% for the CPU and 90% for the Memory). Untapped resources could be used to decrease queuing and consequently turnaround times.

Oracle-based resource shaping We gloss over prediction errors induced by a real statistical model and consider an ideal scenario from the forecasting point of view. Ultimately, our goal is to discern

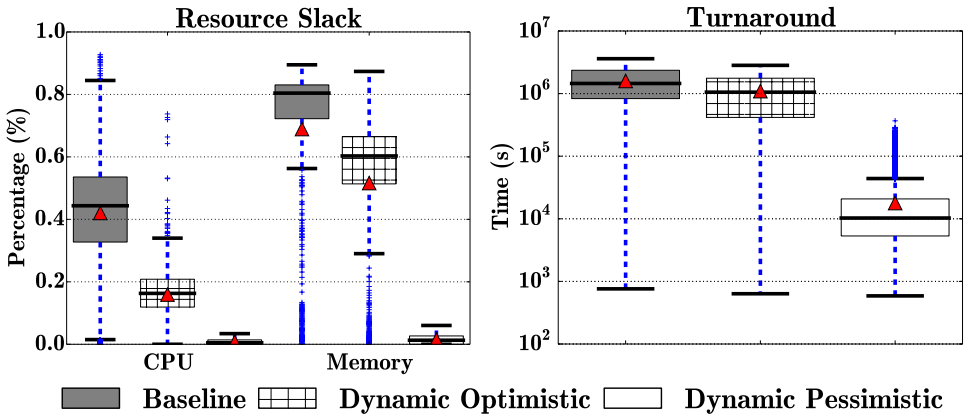


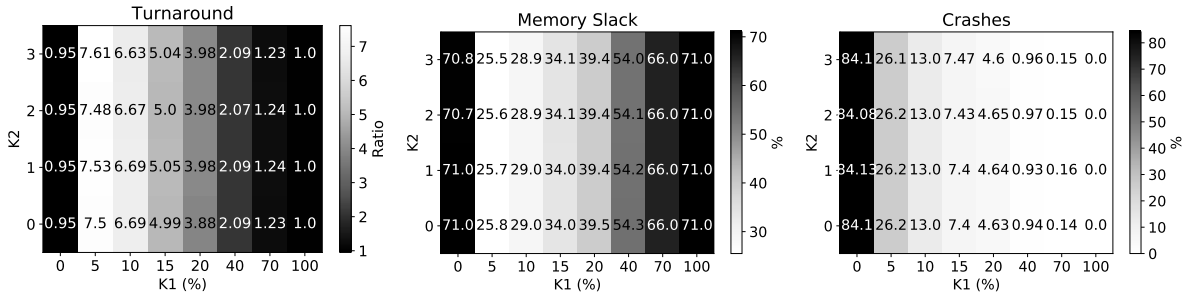
Fig. 4.3 Boxplots comparing baseline vs optimistic vs pessimistic approaches over different metrics, using an oracle in place of the prediction module. The red triangle is the mean.

virtues and drawbacks of different preemption policies. Results are summarized in Figure 4.3: the plots correspond to resources slack and application turnaround, whereas each box correspond to the baseline and our resource shaping approach, with an optimistic (as originally implemented in the Omega simulator [Schwarzkopf et al., 2013]) and our pessimistic preemption policy. Note that our simulator implements the concept of work lost when an application component crashes or gets killed.

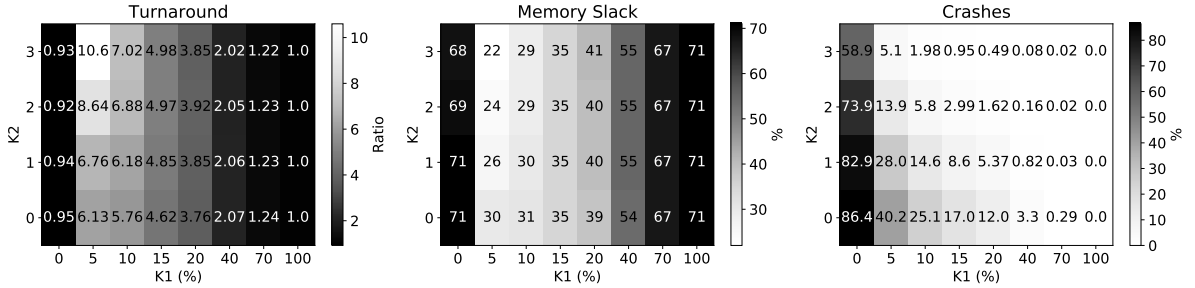
Overall our results indicate that resource shaping brings substantial benefits in terms of all metrics we consider, in the absence of prediction errors. Cluster efficiency improves because resource slack, computed as the difference between allocated and used resources, drastically shrinks as shown in Figure 4.3 (left) compared to the baseline. Similarly, turnaround times are notably smaller as shown in Figure 4.3 (right) in comparison to the baseline. Indeed, the system can ingest new applications more quickly, because resources are better used.

Figure 4.3 can now be used to compare optimistic versus pessimistic eviction policies, in absence of prediction errors. While both approaches improve over the baseline, the pessimistic policy we introduce in this work is consistently superior to the optimistic policy in all respects. As shown in Figure 4.3 (left), the pessimistic policy induces our resource shaping mechanism to follow very closely application resource utilization: in this case, resource slack becomes negligibly small. This result explains why turnaround times, Figure 4.3 (right), are almost two orders of magnitude smaller with the pessimistic policy: by freeing up resources, the application scheduler is amened to trigger new executions, thus queuing times shirk. Furthermore, we compute the number of application failures: in case of the optimistic policy we record 37.67% application failures, whereas with the pessimistic policy no application fails. Indeed, with the optimistic policy, when two applications compete for resources and there are none left, the system will let one of the two fail. Instead, the pessimistic policy avoids failures through partial preemption, by freeing elastic resources first.

ARIMA-based resource shaping Next, we study the system behavior when using ARIMA to predict future resource utilization. As anticipated in Section 4.2, statistical models are prone to prediction



(a) ARIMA



(b) GP

Fig. 4.4 Heat maps showing the effect of K_1 and K_2, which compose β, on different metrics when using ARIMA and GP. Bright cells are better.

errors, which we address using the buffer β. A key feature of our approach is that β is a function of the uncertainty produced by the model. In practice, when the predictor outputs a future (peak) resource utilization, we adjust the value by adding the buffer β. In Figure 4.4a we demonstrate the effect of the buffer parameters ($\beta = f(K_1, K_2)$) on the turnaround ratio over the baseline, the memory slack and application failures (we show average results). In all cases, bright cells are better.

On the x-axis, K_1 controls the static component of Equation (4.3), which gauges the minimum amount of resources systematically granted to applications. The value of K_1 is expressed as a percentage of the requested resources; when $K_1 = 100\%$ our approach degenerates to the baseline. On the y-axis, K_2 controls the dynamic component of Equation (4.3), which integrates prediction uncertainty. We let K_2 vary in the range $[0, 1, 2, 3]$ which define different bands around the mean of the predictive Gaussian distribution, according to the three-sigma rule.

Let's first slice Figure 4.4a by row, and focus on the $K_2 = 0$ case: here we omit uncertainty information and only consider the effects of a static, minimum resource allocation. Even with just $K_1 = 5\%$, our approach achieves 7.5x average improvements in terms of application turnaround, while resource slack is only 30% in average. However, the number of crashed application is high: roughly 26% of applications experience a failure in average, and the situation improves only for large values of K_1. In the limit, when $K_1 = 100\%$, our method degenerates to the baseline: here no application fail, but turnaround times and slack exhibit no improvements. In our system, when an application crashes it is resubmitted and, after a certain amount of failures, the system is not shaping its allocation anymore.

Also, even if applications crash they can still benefit from being able to start sooner than a baseline system because other applications were able to complete their work sooner.

We note that the absence of a static term (i.e. $K_1 = 0\%$) results in turnaround that is very close to the baseline regardless of K_2, due to the high number of applications failures which also lead to an high memory slack. This is a consequence of the occasional high confidence of the predictor in cases where a sudden change in the usage behavior occurs. It is necessary to maintain a static component to accommodate unexpected variations, which are very difficult to capture with statistical methods.

Finally, we focus on $K_1 = 5\%$: the minimum resource allocation is small, and we absorb prediction errors and fluctuations using uncertainty information. However, as K_2 increases, all metrics remain similar: the uncertainty produced by the ARIMA model is not sufficiently accurate to compensate forecasting errors.

GP-based resource shaping. Next, we study the system behavior when using GP regression to predict future resource utilization. Similarly to the ARIMA-based resource shaping, in Figure 4.4b we demonstrate the effect of the buffer parameters. However, we can see that while GP gives slightly worst results when not considering the uncertainty of the forecasting values ($K_2 = 0$) compared to ARIMA, as K_2 increases, all metrics improve: average turnaround ratios increase up to 10.6x improvement, average slack is reduced to a 22% in average, while application failures quickly decrease.

In our setup, the best performance is achieved when the system is most flexible regarding the size of the buffer, i.e., a high value for its dynamic and a small value for its static components.

In summary the results show that, for the best configuration of parameters with a real predictor and not an oracle, tunraround time and resource slack is more than halved in the median case, both in terms of CPU and memory resources. By using the uncertainty provided by the forecasting model based on the GP, we are able to improve these metrics further, achieving 10.6x improvement compared to the baseline for the turnaround time.

4.4 System Implementation

We materialize the ideas presented in this paper with a full-fledged, python-based, implementation of our mechanism, following the system design presented in Section 4.2, and depicted in Figure 4.1. For this work, we build the resource shaper to interact with the application scheduler presented in Section 3.4 [Pace, Venzano, Carra and Michiardi, 2016]. In our implementation, the resource shaper modulates both CPU and memory resources.

In our cluster, we use Docker as the back-end and we have investigated how to resize its containers (corresponding to application components). There are two values that Docker uses to check for Memory limits: a hard and a soft limit. When the hard limit is surpassed, the container is killed by the OS. Instead, when the soft limit is reached, the OS tries to release some resources first. In our work we use the soft limit value since the application scheduler we use takes decisions based on such value. In particular, we rely on the OS low level mechanisms to notify the processes running in the container to

free some of their resources. This practice is compatible with frameworks such as the Java Garbage Collector (GC) that attempts to release allocated but unused memory space. Note that our technique is compatible with approaches such as [Hassan and Zwaenepoel, 2017], which trade performance for a smaller memory footprint.

The monitoring component feeds the utilization forecast module with data at regular time intervals. Frequent updates ultimately result in better system efficiency, as the predictor operates on a high-fidelity view of resource utilization in the cluster. However, this might impose a high toll in terms of monitoring scalability. On the other hand, infrequent updates improve scalability at the expense of lower system efficiency and responsiveness. In our implementation, we collect resource utilization information every minute, which is in line with what done in [Verma et al., 2015].

Next, we provide additional details of our prototype.

Forecasting module It implements the two models we discuss in Section 4.2.1. For the ARIMA model we use the well-known `StatsModel` [Seabold and Perktold, 2010] library, which features an efficient implementation of the ARIMA model and its automatic parameter tuning through the Pyramid wrapper[13]. For the GP model we use the well-known library `GPy`[14]. Both models consider a small history of the ten past observations for training, to keep computational complexity under control.

Resource shaping module It materializes the ideas presented in Section 4.2.2. The ultimate goal of the resource shaper is that of issuing commands to preempt (kill, in our implementation) an entire application, or individual components thereof, and to resize the resource allocation, as computed by the by Algorithm 3. It is important to point out that the resource shaper adapts resource allocations only after enough historical data points are available for the forecasting module: we call this a **grace period**, and set it to 10 minutes in our experiments.

The resource shaper uses the mechanisms exposed by Docker (as discussed above) to adjust application resources, and to eventually preempt components or entire applications. This module computes a new resource allocation for all running application in the system, based on the predicted value and variance obtained from the forecasting module. The buffer β is set to compensate for prediction uncertainty, using the parameters that we obtain through simulations, that is $K_1 = 5\%$ and $K_2 = 3$.

4.5 Experimental Evaluation

We have deployed the mechanism presented in this paper in our cluster, which we operate using [15]. Our goal is to perform a comparative analysis between dynamic resource shaping and a baseline, as done in Section 4.3. The baseline system supports the concept of distributed applications [Pace, Venzano, Carra and Michiardi, 2016], but follows a reservation centric approach, in which allocation matches reservation

[13] https://github.com/tgsmith61591/pyramid
[14] https://sheffieldml.github.io/GPy/
[15] http://zoe-analytics.eu/

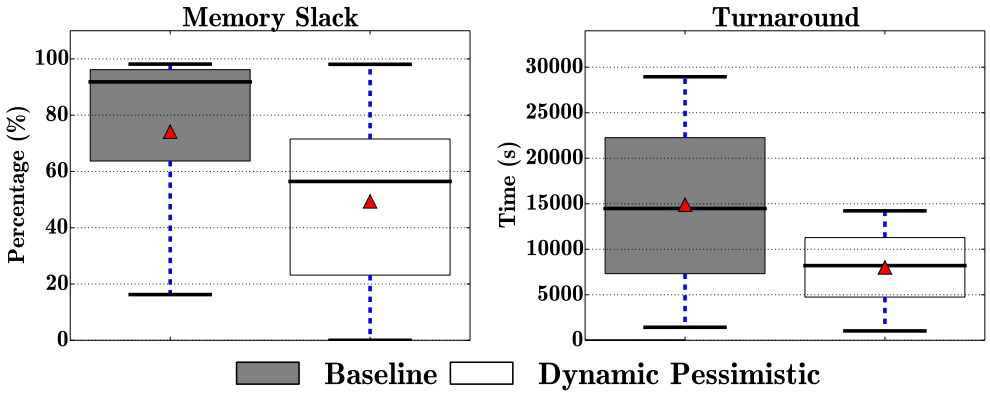


Fig. 4.5 Boxplots comparing baseline vs pessimistic dynamic approach over memory slack and turnaround time distributions using GP-based resource shaping. The red triangle is the mean.

for the entire application lifetime. In our experiments, we consider exactly the same workload trace on both systems which takes approximately 24 hours from the first submission to the completion of the last application.

Workload We use two representative *application templates* including: 1) an elastic application using the Apache Spark framework; 2) a rigid application using the TensorFlow framework. Similarly to the traces used in Section 4.3, we set our workload to include 60% of elastic and 40% of rigid applications, for a total of 100 applications. Application inter-arrival times follow a Gaussian distribution with parameters $\mu = 120$ sec, and $\sigma = 40$ sec, which is compatible with what we observe in our cluster. Regarding the elastic application templates, we consider three use cases. First we consider an application that induces a random-forest regression model to predict flight delays, using publicly available data from the US DoT[16]. Second we consider a music recommender system based on the alternating least squares algorithm, using publicly available data from Last.fm[17]. Third we consider an Extract, Transform and Load (ETL) application. All applications have 3 different flavors: while they all have 3 core components, the number of elastic components varies depending on the flavor. In terms of RAM, all flavors have different reservation values that span from 8GB to 32GB. Instead, using the rigid application template, we train a deep GP model [Cutajar et al., 2017], and use a single TensorFlow instance, with 1 worker and 8-16-32GB of RAM depending on the flavor.

Experimental setup We run our experiment on a isolated platform (which we use as testbed for non-production systems) with ten servers, each with a 8-core CPU running at 2.40GHz, 64GB of memory, 1Gbps Ethernet network fabric and two 1TB hard drives each. The servers use Ubuntu 14.04 and Docker 17.09.0. Docker images for the applications are preloaded on each machine to prevent startup delays and network congestion.

[16] http://stat-computing.org/dataexpo/2009/the-data.html
[17] http://www-etud.iro.umontreal.ca/bergstrj/audioscrobbler_data.html

Summary of results Using the FIFO scheduling policy, and the GP-based utilization forecasting module, we compare the two systems, baseline and dynamic. Overall, the dynamic system is largely more efficient and responsive. We measure substantial improvements in terms of resource allocation: indeed our system can afford to ingest more applications, that would otherwise wait to be served. Figure 4.5 (left) illustrates resource slack, which is roughly 40% lower with our resource shaping mechanism. As a consequence, applications spend less time in the scheduler queue and have short turnaround times, as shown in Figure 4.5 (right). The median turnaround times are $\sim 50\%$ shorter. Note also that the tails of the distributions are in favor of our approach. Finally, we report that no application, nor component failed when using our resource shaping mechanism, configured with the pessimistic preemption policy.

4.6 Summary

The emergence of "the data-center as a computer" paradigm has led to unprecedented advances in cluster management frameworks, that aim at exposing distributed, cluster resources to a variety of business-critical and scientific applications. However, the current resource reservation model hinders an efficient use of cluster resources. Resource utilization dynamics induce over-provisioning, which is one of the main culprit of poor efficiency. The problem of underutilization has been addressed by several approaches. For example, the design of economic incentives to steer system operation has led to the development of complex resource markets, e.g. AWS Spot instances, which call for the design failure tolerant applications, due to the ephemeral nature of the resources they are offered.

In this chapter, we presented a mechanism that cooperates with a scheduler to dynamically adjust resources allocated to an application, so that they closely match those they actually use throughout their lifecycle. Our design featured: a method to build a statistical model to forecast resource utilization, and a preemption policy that reallocates system resources while minimizing failures.

We have validated our mechanism numerically and with a real experimental campaign. Our simulations shed lights on the key role played by our ability to model and use prediction uncertainty, and by the use of strict preemption vs. optimistic concurrency control. We implemented a system prototype of our dynamic allocation mechanism and deployed it in a test environment, where we executed a real workload. Results indicate notably improved system efficiency, which translates in better responsiveness.

Chapter 5

Experimental Evaluation of Disaggregation between Compute and Storage

Nowadays thanks to virtualization, compute and storage clusters are more flexible, they can be easily provisioned in different sizes, and destroyed when not needed[1]. Increasingly, such storage and processing systems are exposed to users as *services*, deployed on either public or private cloud computing environments, rather than on bare-metal machines in private clusters. Indeed, many companies offer Analytics-as-a-Service (AaaS) clusters to run a variety of applications: Amazon Web Services (AWS) with Elastic MapReduce[2], DataBricks Cloud[3], Cloudera Cloud[4] and Google Cloud Hadoop[5] are noteworthy examples.

In cloud computing environments, the architecture of analytics clusters is the result of the composition of several services, consisting of three (logically separated) layers: the *Compute layer* refers to all cluster nodes that run the data processing application (e.g., a Spark application); the *Data layer* refers to any combination of storage services (e.g., HDFS[6] or Swift[7]); and the *Storage layer* that physically stores the data, including ephemeral disks, object and elastic block stores.

In addition, it is likely for the Data or Storage layers and the Compute layer to be on different racks or even data-centers: as a consequence, the traditional wisdom of *data locality* may be challenged. For example, consider Amazon S3[8]: data resides on a set of machines dedicated just to storage, breaking data locality completely.

[1] https://aws.amazon.com/application-hosting/benefits/
[2] https://aws.amazon.com/emr/
[3] Solution hosted on AWS: https://databricks.com/product/databricks-cloud
[4] Solution hosted on AWS: http://www.cloudera.com/content/cloudera/en/solutions/partner/Amazon-Web-Services.html
[5] https://cloud.google.com/hadoop/
[6] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[7] http://docs.openstack.org/developer/swift/development_saio.html
[8] https://aws.amazon.com/s3/

Currently, users of AaaS have abundant information about pricing and about the durability of resources. It is possible to reason about cost-based service dimensioning, and to select appropriate storage services depending on data availability and durability objectives. As a consequence, it is today possible to build data ingestion, storage and processing pipelines, by composing – in various combinations – the three layers defined above.

The questions that we address in this chapter then is: what happens to the performance, and to the completion time in particular, of analytics applications with different type of Compute and Data layer configurations?

We take an experimental approach, and propose a measurement methodology and campaign, whose objective is to analyze the performance corresponding to an intuitive notion of distance between where computation happens and data reside. In doing so, we define an extensive set of application workloads that challenge the systems under study in different ways. Ultimately, our goal is to overcome the limitations of prior works that only provide a boolean vision of data locality: our results indicate that – in general – the intuitive distance metric we present in this work is a good proxy to reason about performance ranking. However, impedance mismatch between different services and application workloads must be taken into account to formulate plausible explanations for outliers in terms of performance.

In particular we show that data locality cannot be studied as a feature that can be either present or not, as it happens nowadays, but that there exist different degrees of data locality, whose importance varies depending on the configuration of the data analytics framework and on the specific workload. Furthermore we point out some design problems of a specific storage service architecture (Swift) making this solution non-optimal for running a data analytics framework; and we show the impact of caching *hot* data at the Data layer level with respect to application runtime.

This chapter is divided into different parts: Section 5.1 explain our research question, Section 5.2 describes the methodology and Section 5.3 is dedicated to the results.

5.1 Problem Statement

We empirically evaluate the performance of analytics applications composed using a variety of Compute, Data and Storage layer configurations. Our goal is to understand how application run time varies across configurations, for a wide range of application workloads.

Performance modeling of complex, distributed systems is a daunting task: application runtime is affected by several factors, including *data locality* (which is the foundation of parallel processing frameworks such as Hadoop and Spark), impedance mismatch between the various services involved in analytic applications, interference between competing tenants, application workloads, and many more.

As such, we take an experimental approach, and analyze application performance through the lenses of the data locality principle, which we revisit to accommodate the breath of storage configurations currently available in most public and private clouds. In our study, we emphasize problems that arise as a consequence of service composition, and suggests ways to mitigate them.

5.2 Methodology

We use a private cloud computing platform to overcome the limitations of experiments performed on public cloud infrastructures and we run 4 different types of analytics applications: read intensive, write intensive, business intelligence and machine learning. In this section we: *(i)* provide the specifics of our platform, *(ii)* illustrate the different placements of Compute, Data and Storage layers that we use in our experiments, *(iii)* introduce an intuitive notion of distance between where computation happens and data reside, that we call the *Compute-to-Data path*, *(iv)* present our workloads and *(v)* explain the metrics used.

5.2.1 Experimental Platform

Our platform is composed by 25 server-grade machines equipped with: *(i)* two sockets with an Intel Xeon at 2.40GHz, with hyper threading enabled (32 Cores), *(ii)* 128 GB RAM and *(iii)* ten 7200 RPM 1 TB disks. The platform is distributed across three racks. All the switches in the network topology can be considered "non-blocking": all machines can communicate at 1Gbps with each other.

We operate our platform using OpenStack[9], which can automatically provision virtual analytic clusters composed by Virtual Machines (VMs) connected directly to each other (i.e., no traffic encapsulation and no virtual routing).

Our platform provides volumes (similar to Amazon EBS[10]) and ephemeral disks Storage layers. Volumes are provisioned through the Openstack's Cinder[11] module on top of Ceph [Weil et al., 2006] that is a distributed file system featuring high performance, reliability and scalability. Ceph's blocks are distributed over 8 disks spread across 5 physical machines. Similarly to a traditional RAID 0 approach, when performing read and write operations, Ceph divides the data in smaller chunks (8 MB) and store them across storage servers, called Object Storage Daemons (OSDs). Instead, ephemeral disks are connected to a portion of a single physical disk that reside on the same host that runs the virtual machine using it.

Our private cloud uses the OpenStack Sahara project to automatically provision Compute layers: in this work, we use Apache Spark[12]. Spark can read from several Data layers; the two widespread solutions that we take into consideration are HDFS and Swift (which is similar to Amazon S3[13]). HDFS is a Java-based distributed file system providing scalable and reliable data storage, designed to store a small number of large files. Swift is a highly available, distributed, eventually consistent object store designed as a generic service to reliably store very large numbers heterogeneous files. In our platform, Swift is deployed on a single physical machine using the Swift-All-in-One (SAIO[14]) configuration, on the basis of the capacity planning suggested in [Arnold, 2014]; no other processes share Swift's hardware.

[9] http://openstack.org
[10] https://aws.amazon.com/ebs/
[11] http://docs.openstack.org/developer/cinder/
[12] http://spark.apache.org/
[13] https://aws.amazon.com/s3/
[14] http://docs.openstack.org/developer/swift/development_saio.html

More generally, Swift can be deployed on several machines, to increase, e.g., capacity and reliability, at the cost of increased network traffic. It is worth noting that we disabled the Swift authentication mechanism in order to avoid additional overhead in the communication process, and to focus only on the data path.

In this work, we gloss over the intricacies due to multi-tenancy and interference: hence, we statically allocate a portion of the platform to run our experiments. The Compute layer is virtualized on 5 VMs and uses Spark 1.5.2 as the computing framework: 4 workers, spread across 4 different hosts, and 1 master. The Data layer is also virtualized in 5 VMs spread across 5 different hosts. All the VMs are equipped with *(i)* 4 cores, *(ii)* 32 GB of RAM and *(iii)* 80 GB of disk.

To gain statistical confidence in our results, all the experiments we report in this article were repeated five times.

5.2.2 Deployment scenarios

We define a deployment scenario as a configuration of Compute, Data and Storage layers and study 4 scenarios that we think representative of common configurations:

Guest Collocation (GC) The Compute and Data layers are hosted on the same VM, the Storage layer is an ephemeral, local disk. This is a popular configuration in public clouds: when a GC cluster is decommissioned, all data is lost.

Guest Collocation with Volumes (GC-V) Same configuration as for GC, but the Storage layer uses volumes provisioned using the Ceph distributed file system. This is also a popular configuration in public clouds: it enables elasticity at the Compute layer, without sacrificing data durability at the Data and Storage layers.

Swift (SWI) The Compute and Data layers run on distinct hosts. The Data layer is the Swift object store, which ensures data durability. Similarly to the GC-V configuration, this scenario enables Compute layer elasticity, and it is a popular configuration for its simple REST-based interface to interact with data.

No Collocation (NC) The Compute and Data layers run on different hosts; the Data layer uses HDFS mounted on a Storage layer that uses ephemeral, local disk. This is a scenario enabling data durability: the Compute layer can be decommissioned, while the Data and Storage layer keep running.

As we discuss in section 5.2.3, each scenario presents a different degree of data locality. In addition, we note a first example of impedance mismatch, which we further elaborate in the remainder of this work. Indeed, both the Data and Storage layers might implement their own data replication mechanism. This is evident in the GC-V scenario: when HDFS is mounted on volumes, HDFS and Ceph replication mechanisms are redundant. To better understand the performance implications of the GC-V scenario, we thus distinguish two cases: GC-Vb, with both HDFS and Ceph replication and GC-Vs, with only

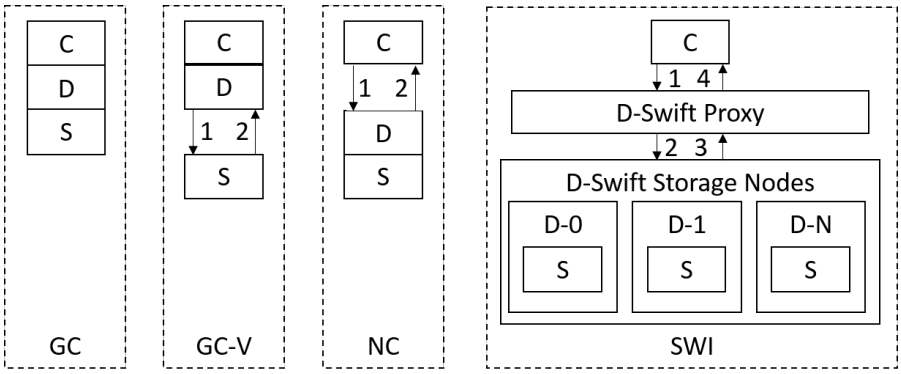


Fig. 5.1 *Compute-to-Data path* for different scenarios during read operations.

Ceph replication, which correspond to the degrees of freedom exposed to users for configuring their services. Note that if HDFS replication is disabled, Spark (the Compute layer) has no other means to retrieve the data if a datanode crashes, resulting in a failure of the application; also Spark's scheduler has less flexibility in scheduling tasks, because data blocks are present in only one datanode. In contrast, by enabling HDFS replication, the application will write extra data.

5.2.3 Compute-to-Data path

We now introduce an intuitive notion of distance between where computation happens and data reside. As illustrative examples, consider the following cases: compute and data reside on the same VM, on different VMs running in the same physical host, on different VMs on different physical hosts in the same rack, and so on. It is intuitive to treat these cases as increasing in terms of distance, which is thus loosely coupled with the amount of network links data need to traverse for being processed. Also, recall that read and write operations issued by the Compute layer, work on a given split of the input or output data, which is organized as a sequence of records. We use the following intuitive and rough definition of distance:

Definition *The Compute-to-Data path is the number of logical links a successful (read or write) operation must cross, for a given data record.*

In this Section we use the *Compute-to-Data path* as a proxy to reason about performance ranking, that takes into account the logical distance between the three layers composing an analytic service and the additional cost of the replication system(s). Indeed, we can expect a performance degradation each time an operation traverses a network link: the intuitive ranking holds even if we do not explicitly model network latency or topology.

Figure 5.1 shows graphically how we derive the *Compute-to-Data path* for each scenario during a **read request**. Note that it is important to be careful and take into account the architecture details of each layer: for example, Swift has a single point of access called the Swift-Proxy, which mediates between the Compute layer and Swift's storage nodes. To calculate the *Compute-to-Data path* we count all the

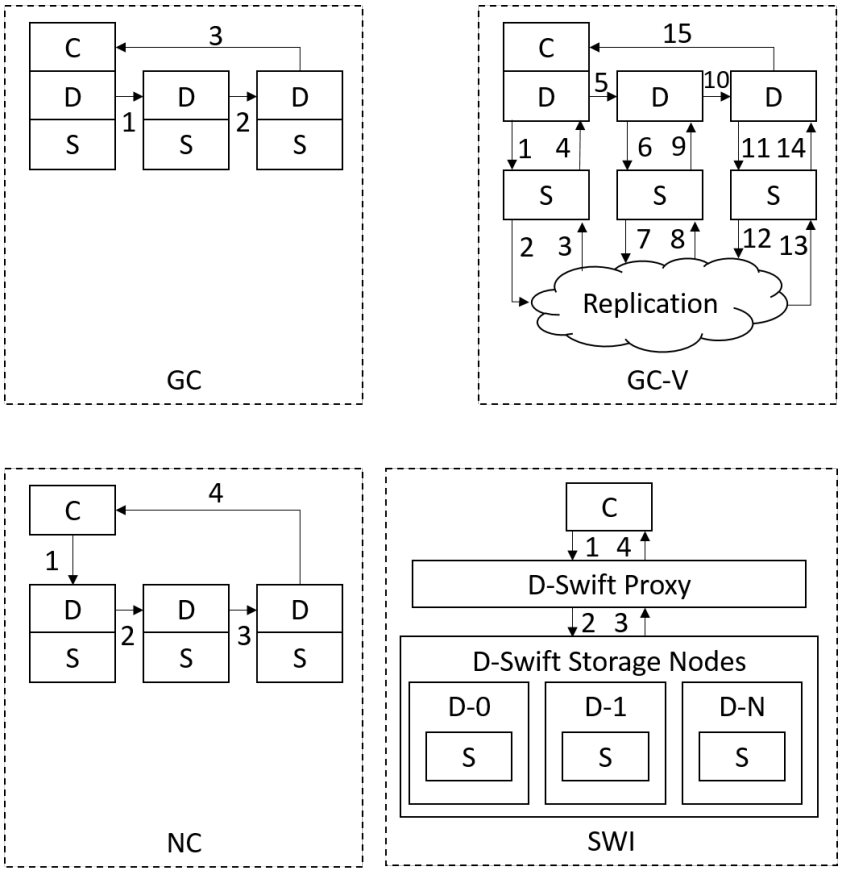


Fig. 5.2 *Compute-to-Data path* for different scenarios during write operations.

logical links, between the Compute layer and the physical data, that each individual record request has to traverse. Considering the SWI scenario, we have one link from Compute layer to Swift-proxy, then one more between Swift-proxy and Swift' storage nodes, finally the record request's ACKs traverse the same links to reach the Compute layer, for a total of 4 links. Similar consideration can be made for the remaining scenarios: the GC traverses 0 link while the GC-V and the NC 2 links.

For **write requests**, data replication has to be taken into account. HDFS, Swift and Ceph have different replication systems: HDFS uses *chain-replication*, whereas Swift and Ceph use *asynchronous replication*, with different quorums. Assuming a replication factor of 3, the Swift-proxy requires 2/3 of storage nodes to acknowledge a write operation, whereas Ceph requires all OSDs to acknowledge success. Figure 5.2 illustrates how to derive the *Compute-to-Data path* for each scenario during a write request. Taking GC-Vb as example: a single datanode record write operation traverse 4 links, since we have HDFS replication active with a factor of 3, we will have 12 links; we also have to count the links between datanodes and the final ACK, for a total of 15 links.

The *Compute-to-Data path* for read and write operations, and for different scenarios is summarized in table 5.1. In the Table, we organize and rank scenarios based on their distance: intuitively, we expect application performance to follow the same ranking we produce using the *Compute-to-Data path*. Our

Table 5.1 Expected scenarios' performance ranking.

(a) Read

Rank	Scenario	#links
1	GC	0
2	GC-V	2
2	NC	2
3	SWI	4

(b) Write

Rank	Scenario	#links
1	GC	3
2	GC-Vs	4
2	NC	4
2	SWI	4
3	GC-Vb	15

measurement results indicate that the *Compute-to-Data path* is a good proxy to rank scenarios based on their expected relative performance, albeit intuition is not sufficient alone to explain what we support with data.

5.2.4 Benchmark and Workloads

To study the performance of analytics applications we use four workloads (described in details in table 5.2), that are currently used in popular benchmark tools suites and cover different kinds of applications. We use WordCount and DFSIO from Intel Hi-Bench[15] [Huang et al., 2010] test suites; the first is the "Hello World" application for parallel computing, which is a read-intensive workload, while the second is a write intensive application: they both perform read and write operations on plain-text files. TPC-DS is a transaction processing performance Council's decision-support benchmark test[16] [Nambiar and Poess, 2006], by DataBricks' Spark-Sql-Perf library[17], that executes 5 complex queries[18] from files stored using the Parquet Format[19]. Decision-Tree is a machine learning algorithm taken from Spark's MLlib library[20] [Meng et al., 2016] that reads CSV files and builds a statistical model of the underlying data distribution; this is the only workload that uses *caching* for the input data: due to its iterative nature, this is the current best-practice to achieve low training times.

5.2.5 Performance metrics

To investigate the impact that different configurations have on application performance, we use 4 metrics. The first is the *job runtime*: this is the amount of time required by the application to terminate its execution. To delve into the reasons behind each workload's behavior in each scenario, we define extra

[15] https://github.com/intel-hadoop/HiBench
[16] http://www.tpc.org/tpcds/
[17] https://github.com/databricks/spark-sql-perf
[18] In Databricks' library they are called *simple-queries*.
[19] https://parquet.apache.org/
[20] http://spark.apache.org/docs/latest/mllib-guide.html

Table 5.2 Workloads' details.

Workload	#Jobs	#Mapper	#Reducer	Input Size	Output Size
WordCount	1	158	158	20 GB	225.6 MB
DFSIO	1	160	0	0 B	20 GB
TPC-DS	217	16160	16821	13 GB	17.9 KB
Decision-Tree	153	4549	4825	3 GB	8 MB

metrics collected for each analytics application during its execution. These metrics are the percentages of CPU, Network and Disk used by the application itself, computed by standard tools such as *iostat*[21].

To compute the above metrics, we monitor each component of the clusters deployed for a specific scenario: Spark master, Spark worker, HDFS namenode, HDFS datanode and Swift[22]

5.3 Results

In the following we analyze the performance of each workload and its behavior on each scenario, and summarize our findings, discussing the implications for both end-users and providers of cloud services. Furthermore, we discuss about possible directions to mitigate the performance degradation that some Data and Storage layers incur.

5.3.1 Analytics Application Benchmark

Application performance is easier to understand when results are grouped by workload type. For this reason, in what follows we first delve into the details of each application we use in our experiments. The template we use in our analysis is as follows: first, we discuss whether the ranking produced by our intuitive *Compute-to-Data path* matches that of real workloads, then provide experimental evidence to explain outliers[23].

WordCount

In general, we remark that the expected rank produced in table 5.1 is representative of application performance: from table 5.3a we see that the GC and GC-V scenarios have roughly the same performance, whereas the NC and SWI scenarios are slower.

The NC scenario constitutes an interesting out-lier: application runtime is roughly 25% slower compared to higher rank scenarios. This is caused by a low CPU utilization that is a direct effect of an inefficient use of network resources: indeed, all read/write operations are synchronous, thus the CPU is blocked until the operation is completed. Note that, although the NC and GC-V cases have the same

[21] http://linux.die.net/man/1/iostat
[22] We monitor the Swift-All-in-One deployment as a single component.
[23] Not all figures will be shown due to space constraint.

Table 5.3 Analytics applications benchmark results in ascending order.

(a) WordCount

Rank	Scenario	Run Time (s)
1	GC-Vb	121.29 ± 2.20
1	GC	125.23 ± 2.15
1	GC-Vs	125.28 ± 1.89
2	NC	157.85 ± 2.94
3	SWI	279.55 ± 4.07

(b) TPC-DS

Rank	Scenario	Run Time (s)
1	GC-Vb	454.48 ± 6.89
1	GC	460.21 ± 3.95
1	GC-Vs	469.66 ± 9.31
2	NC	571.01 ± 3.98
3	SWI	2773.96 ± 16.89

(c) DFSIO

Rank	Scenario	Run Time (s)
1	GC	305.86 ± 14.68
1	NC	308.83 ± 13.38
1	GC-Vs	330.99 ± 13.15
2	GC-Vb	848.48 ± 60.74
3	SWI	1114.56 ± 28.22

(d) Decision Tree

Rank	Scenario	Run Time (s)
1	GC-Vb	997.50 ± 16.47
2	NC	1067.35 ± 33.48
2	GC	1076.68 ± 39.74
2	SWI	1101.37 ± 21.74
2	GC-Vs	1133.56 ± 37.13

Compute-to-Data path, their data access mechanism is different. In the NC scenario, when a Compute instance requests a record from a datanode, the record is read over a single disk and network link, which performs poorly overall. Furthermore, aside from being a slow configuration, the NC scenario is also the most expensive one.

Instead, the GC-V scenario achieves very good performance, even if the network cards we use in our platform (1 Gbps interfaces) are not on par with what is currently deployed in public clouds such as AWS (10 Gbps interfaces). This is the result of parallel data transfers, which use network resources more efficiently: fragments of data records are read or written by several disks and transfered over multiple network links. As such, the expected performance degradation caused by a large *Compute-to-Data path* is practically nullified. Additionally, we remark that application performance can reap the benefits of having both Data and Storage layer replication enabled (GC-Vb): indeed, Spark's scheduler has more flexibility in choosing the designated executor for a specific task since there are multiple copies of the same data block.

As expected, the SWI configuration achieves the worst performance: as we discuss later, this is due to both poor architectural choices (Swift was not originally designed to serve parallel processing frameworks) and to problems that arise between Spark and Swift.

TPC-DS

Table 5.3b indicates that there is a good match between the expected ranking obtained using the *Compute-to-Data path* (TPC-DS is a read intensive workload) and application runtimes. Similarly to the WordCount workload, the NC scenario suffers from inefficient use of network resources. Figure 5.4, which shows the CDF of the task runtime for each workload, supports this claim: tasks are slower in the NC configuration, compared to the GC and GC-V scenarios.

To better understand the causes of the increased tasks runtime, in the SWI scenario, we study resource utilization. Figure 5.3 shows the resource utilization across different scenarios. In SWI, the network may be considered the source of slowdown (as shown by the median at almost 100%): in fact, in Swift all requests to read or write data records pass through the same physical channel (at the proxy), stressing the network and its chances to become a bottleneck. To tackle this issue, AaaS providers adopt two approaches: faster network links (e.g., 10 Gbps) and deployment of several proxies to balance the traffic. These solutions partially mitigate the problem. Using faster network links may work, but does not scale with the number of concurrent tenants. Using several proxies and a load balancer introduces other problems: *(i)* additional delays due to an extra communication step; *(ii)* a single storage node will have to handle more requests coming from different proxies, thus moving the pressure from the link *Compute-to-Proxy* to the link *Proxy-to-Storage nodes*. Inevitably, additional proxies bring Swift architecture closer to that of HDFS-like system, but at an higher cost in terms of required hardware and consequently, requiring a difficult capacity planning. Additional factors that contribute to performance slowdown include the extra time required by Swift to parse HTTP requests and to dispatch them to the different storage nodes. Finally, from fig. 5.3 we can also see how the CPU utilization in the Compute layer of the SWI scenario is lower compared to the other scenarios; similarly to WordCount, this is because the Compute layer has to wait more time to process the data.

Clearly, a narrow measurement campaign that focuses on a single scenario (e.g., the GC configuration) might lead to inaccurate conclusions: even if the workload may be considered CPU-bound – thus suggesting data locality to be irrelevant – different configurations with different levels of data locality have a non-negligible impact on application runtimes.

DFSIO

Table 5.3c shows the measured runtime of DFSIO, which is a write intensive workload, for several configurations. As expected, the intuitive ranking we derive using the *Compute-to-Data path* only partially matches with experimental results: impedance mismatch between layers is the main culprit for performance degradation.

First, we focus on the SWI scenario: to better understand our experimental results we run a micro benchmark that, using the *python-swift*[24] client, emulates the DFSIO workload: writing the same amount of data as for the DFSIO workload takes only 300 seconds. A detailed log inspection indicates that Swift – because it stores immutable objects with immutable identifiers – does not work well in conjunction with

[24] https://github.com/openstack/python-swiftclient

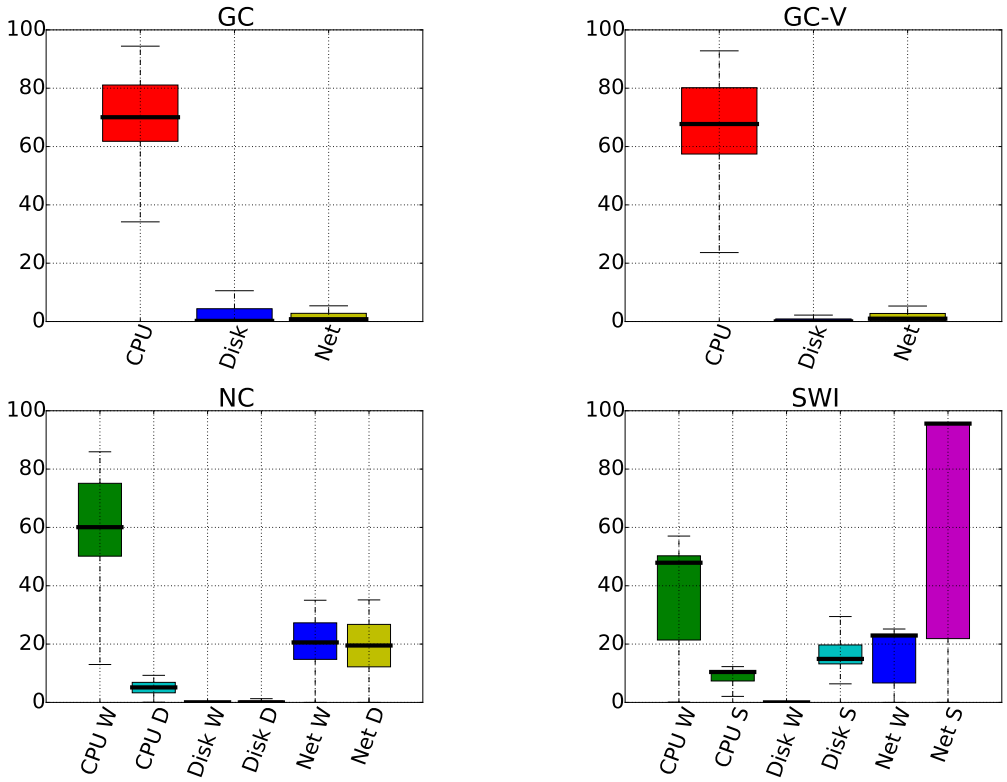


Fig. 5.3 Resource utilization with the TPC-DS workload in different scenarios. The ticks on the X axis "W", "D" and "S" stand, respectively for worker, datanode and Swift machines. The resource utilization reported is a global average across all instances of each layer. The network utilization between the GC-V and GC scenarios is similar because volumes' network is opaque to the Operating System and, therefore, counted in the disk utilization.

current parallel processing frameworks such as Spark. Indeed, Spark tasks always output temporary files that are renamed once the processing is complete[25]. Since in Swift a **rename** operation is implemented as a **copy** operation, the DFSIO workload in the SWI configuration involves writing much more data than required, hence the poor performance. In particular the data are written 2 extra times, the first from tasks themselves and a second time from the Spark master when the job is completed. From the logs we can see that the job wrote 20 GB in 600 seconds (s) while the application ran in 1114s; in the 600s spent by the job, every task performed a rename operation writing the output one extra time; the difference between the job and the application runtime (514s) corresponds to the time spent by the Spark's master to rename all the files written by the tasks, that is, to write the output a third time. Figure 5.4 shows the distribution of task runtimes for the DFSIO workload: for the SWI scenario, tasks are much slower than in other scenarios, which corroborates our claims[26].

Next, we focus on the GC-V scenarios. In this case, the impedance mismatch between Data and Storage layers is to blame for poor workload performance. Table 5.3c indicates that by disabling HDFS

[25]This is reminiscent of the failure tolerance mechanisms in Spark, that assumes tasks can fail at any time.
[26]We are aware of a work from IBM (https://github.com/SparkTC/stocator) that aims to address the file renaming problem.

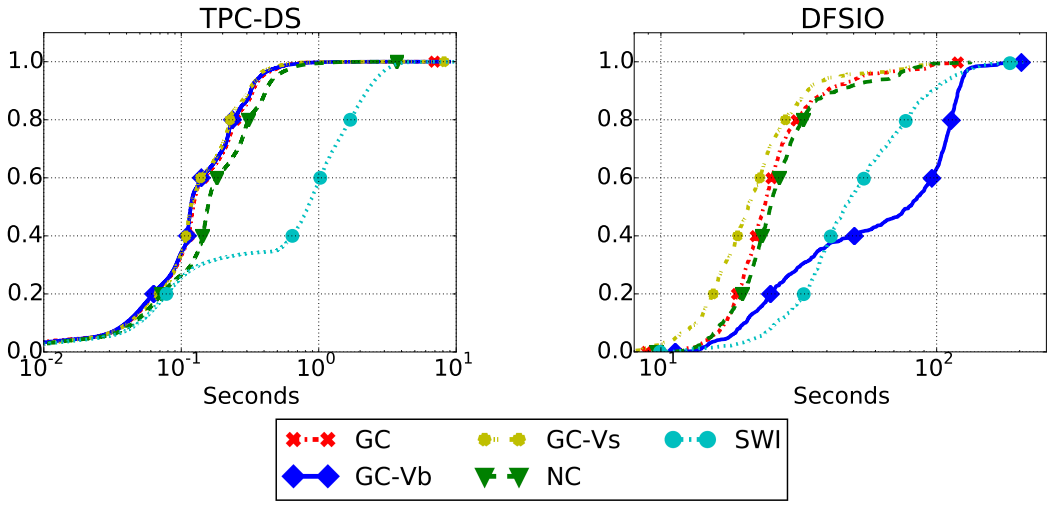


Fig. 5.4 DFSIO and TPC-DS CDFs for task runtimes in all scenarios.

replication mechanism (scenario GC-Vs), performance drastically improves when compared to a naive deployment with both replication mechanisms (HDFS and Ceph) enabled. Although this is not possible to study in a public cloud, we also run experiments in which we disable Ceph replication and only use HDFS replication: in this case, the application runtime falls to roughly 250 seconds, that it is 20% faster to complete with respect to the GC scenario. In summary, write-intensive applications can be significantly affected by impedance mismatch between services: replication and failure tolerance mechanisms implemented in different layers must be tuned to achieve better performance.

Decision-Tree

The decision tree algorithm taken from Spark's MLlib is at the heart of our machine learning use case: it is an iterative algorithm that builds a statistical model using millions of training data points. Table 5.3d indicates that application runtimes are essentially independent from system configurations. This is the results of using Spark caching mechanism. When using caching at the application level, the interaction between Compute and Data/Storage layers is limited to reading the input data; the bulk of the computation, and hence the application runtime, happens during the iterations of the decision tree algorithm, in which the Data and Storage layers do not intervene. Finally, the output of the algorithm is a statistical model, which is very small for this workload.

Finally, we note that the GC-Vb scenario performs better than others: indeed, *(i)* Ceph reads from multiple disks and *(ii)* Spark's internal scheduler has more flexibility in placing tasks because of the increased data redundancy at the HDFS layer. In general, the most used resource is CPU, while network and disk I/O are barely used.

5.3.2 Summary of the results

Choosing the right composition of analytic services is a difficult problem, involving cost considerations, data durability requirements, and ultimately, expected application performance. Our experimental findings pave the way to informed decisions about AaaS deployments. In the following, we summarize our results and their implications.

Service composition A configuration that aims at achieving data durability in spite of the ephemeral nature of VMs and the services they execute, must be designed with care. For reasons ranging from ease of integration to familiarity with well-established APIs, it is tempting to compose services as done in the NC (no collocation) scenario we study in this work. Our results show that this is a bad choice for a wide range of workloads, in which precious CPU cycles are lost to wait for data to travel over the network.

Volumes Using volumes provisioned on top of a distributed file system like Ceph perform surprisingly well. This is unexpected as, similarly to the NC scenario, the network is heavily involved during application execution. However, our results indicate that even with a modest bisection bandwidth, the Compute layer can make quick progress toward the end of an application, thanks to the efficiency of striping.

However, as cautionary note, our results indicate a potential impedance mismatch between Data and Storage layers, due to the interaction of multiple replication mechanisms. As such, end-users should be aware of the situation, and appropriately configure the Data layer, such that data replication is only performed by the Storage layer, because this is of great benefit to application performance.

Additionally, our results indicate that cloud computing providers could differentiate their volume offering: general purpose volumes would work as usual, whereas analytics volumes should disable data replication. In this case, end-users would be in complete control of replication: our results show that – especially for write intensive workloads – this produces superior performance.

Swift The performance of Swift is disappointing. This is due to another instance of impedance mismatch, between the Compute and Data layer. Swift inability to rename files without actually creating a new copy, causes severe performance penalties, making Swift a sub-optimal solution.

In addition, we note that the Swift architecture was designed for applications that are very different from parallel computing frameworks: our results indicate that the Swift proxy may represent a bottleneck, since it is involved in the data path. Certainly, using a proxy server as coordinator enables cluster managers to easily add control flows to Swift, but this degrades performance. One solutions that is currently adopted by several companies using Swift, at a production level, is to add several proxies and balance the traffic load between them: but because the workload may change unpredictably, a well thought capacity plan is not easy to obtain. As previously underlined, more proxies will make the Swift's architecture somehow similar to HDFS, but at higher costs. Recent work from IBM[Rabinovici-Cohen et al., 2014] shows that some control flow and data transformations can be done much closer to the

storage nodes. As a consequence, it is tempting to suggest the design of a new Swift proxy that could behave similarly to the HDFS NameNode: such alternative proxy would only act as a metadata storage, and would not be involved in the actual data path.

Caching Finally, our results show that caching plays an important role in determining application performance. On the one hand, caching "breaks" the *Compute-to-Data path* that can be inferred from read/write operations on data records, which makes application performance more difficult to predict. On the other hand, by collapsing the *Compute-to-Data path*, it mitigates the problems of several configurations we studied, which is helpful for end-users because it gives more flexibility in choosing Data and Storage layers.

However, the design of inter-application caching mechanism for parallel processing frameworks is still in its infancy: Tachyon [Li et al., 2013] and HDFS2 are good examples of recent approaches to tackle this problem.

5.4 Summary

We investigated the impact of different Compute, Data and Storage layer configurations on the performance of a data analytic framework. We took an experimental approach, and proposed a measurement campaign, whose objective was to analyze workload performance in light of an intuitive notion of distance between where computation happens and data reside. First, we discussed how to approximately rank different service compositions, in terms of expected performance. Then we performed an extensive measurement campaign on a private cloud computing environment. Results indicated that, in general, our intuitive distance metric is a good proxy to reason about performance ranking. Finally we presented experimental evidence of the impedance mismatch that affect two important storage layers – object and elastic block stores – and deduced mechanism to mitigate negative effects on performance.

Chapter 6

Stocator: High Performance Connector for Object Stores

In this chapter we present Stocator[1], a high performance storage connector, that enables Hadoop-based analytics engines to work directly on data stored in object storage systems. Here we focus on Spark however, our work can be extended to work with the other parts of the Hadoop ecosystem.

Until now Hadoop connectors to object storage, e.g., S3a[2] and the Hadoop Swift Connector[3], have been based on file semantics, a natural assumption given that their model of operation is based on the way that Hadoop interacts with its original storage system, Hadoop Distributed File System (HDFS). However, treating object storage like a file system constitutes an impedance mismatch, which can lead to poor performance and incorrect execution. In particular, operations that are atomic for files may not be atomic for objects and operations that are inexpensive for files may not be inexpensive for objects, and vice versa. For example, to rename a directory in a file system requires a single atomic operation, whereas in object storage it requires copy and delete operations for each of the objects in the tree under the "virtual directory"[4].

We are not the first to recognize the poor performance of the object storage connectors. Others have tried to improve performance, by sacrificing speculative execution, and then writing objects directly to their final names, e.g., the DirectOutputCommitter[5] for Amazon S3, or by renaming Hadoop output objects to their final names when tasks complete (task commit) instead of waiting until the entire job completes (job commit)[6]. However, due to the impedance mismatch these attempts led to subtle failures.

Current connectors can also lead to failures and incorrect execution because the list operation on object storage containers/buckets is eventually consistent. EMRFS[7] from Amazon and S3mper[8] from

[1] https://github.com/CODAIT/stocator
[2] https://aws.amazon.com/sdk-for-java/
[3] https://github.com/openstack/sahara-extra/tree/master/hadoop-swiftfs
[4] Object stores emulate directories through hierarchical naming.
[5] https://github.com/apache/spark/pull/12229
[6] https://issues.apache.org/jira/browse/MAPREDUCE-6336
[7] https://aws.amazon.com/blogs/aws/emr-consistent-file-system/
[8] http://techblog.netflix.com/2014/01/s3mper-consistency-in-cloud.html

Netflix overcome eventual consistency by storing file metadata in DynamoDB[9], an additional strongly consistent storage system separate from the object store. A similar feature called S3Guard[10] is being developed by the Hadoop open source community for the S3a connector. Solutions like these, which require multiple storage systems, are complex and can introduce issues of consistency between the stores. They also add cost since users must pay for the additional strongly consistent storage.

The novel algorithms of Stocator achieve both high performance and fault tolerance by taking advantage of object storage semantics. This greatly decreases the number of operations on object storage as well as enabling a much simpler approach to dealing with the eventually consistent semantics typical of object storage. We have implemented our connector for both the OpenStack Swift API[11] and the Amazon S3 API, and have shared it in open source[12]. We have compared its performance with the S3a and Hadoop Swift connectors over a range of workloads and found that it executes far less operations on the object store, in some cases as little as one thirtieth of the operations. Since the price for an object storage service typically includes charges based on the number of operations executed, this reduction in the number of operations lowers costs in addition to reducing the load on client software. It also reduces costs and load for the object storage provider since it can serve more clients with the same amount of processing power. Stocator also substantially increases performance for Spark workloads running over object storage, especially for write intensive workloads, where it is as much as 18 times faster. Stocator is in production in the IBM Cloud and has enabled the SETI project to perform computationally intensive Spark workloads on multi-terabyte binary signal files[13].

The remainder of this chapter is structured as follows. In Section 6.1 we present background on Apache Spark and we motivate our work. In Section 6.3 we describe how Stocator works. In section 6.4 we present the methodology for our performance evaluation, including our experimental set up and a description of our workloads. Finally, in section 6.5 we present a detailed evaluation of Stocator, comparing its performance with existing Hadoop object storage connectors, from the point of view of run time, number of operations and resource utilization.

6.1 Apache Spark

Spark can read from several Data layers; the two widespread solutions that we take into consideration are HDFS and Swift. The first is a Java-based file system providing scalable and reliable data storage, designed to store a medium number of large files to support data processing. The second is a highly available, distributed, eventually consistent object store designed as a more generic storage solution to reliably store very large numbers of different sizes' files.

We describe Apache Spark's execution model and how it interacts with storage, pointing out some of the problems that arise when it works on data in object storage.

[9] https://aws.amazon.com/dynamodb/
[10] http://www.slideshare.net/hortonworks/s3guard-whats-in-your-consistency-model
[11] https://developer.openstack.org/api-ref/object-storage/
[12] https://github.com/SparkTC/stocator
[13] https://medium.com/ibm-watson-data-lab/simulating-e-t-e34f4fa7a4f0

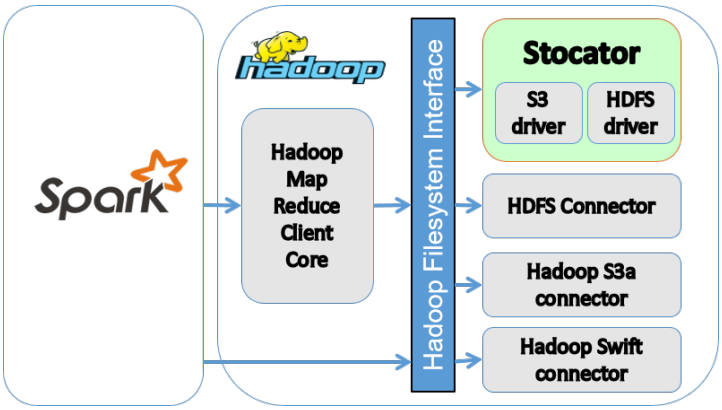


Fig. 6.1 Hadoop Storage Connectors

Spark execution model The execution of a Spark application is orchestrated by the *driver*. The driver divides the application into *jobs* and jobs into *stages*. One stage does not begin execution until the previous stage has completed. Stages consists of *tasks*, where each task is totally independent of the other tasks in that stage, so that the tasks can be executed in parallel. The output of one stage is typically passed as the input to the next stage, so that a task reads its input from the output of the previous stage and/or from storage. Similarly, a task writes its output to the next stage and/or to storage. The driver creates worker processes called *executors* to which it assigns the execution of the tasks.

The execution of a task may fail. To overcome a failure the driver starts a new execution of the same task. The execution of a task may also be slow and the driver may not be able to tell whether the execution has failed or is just slow. Spark has an important feature to deal with slow execution called *speculation*, where it speculatively executes multiple executions of the same task in parallel. Speculation can cut down on the total elapsed time for a Spark application/job. Thus, a task may be executed multiple times due to a failure or speculation and each such *attempt* to execute a task is assigned a unique identifier, containing a job identifier, a task identifier and an execution attempt number.

Spark and its underlying storage Spark interacts with its storage system through Hadoop, primarily through a component called the Hadoop Map Reduce Client Core (HMRCC) as shown in the diagram on the left side in fig. 6.1. HMRCC interacts with its underlying storage through the Hadoop File System Interface. A connector that implements the interface must be implemented for each underlying storage system. For example, the Hadoop distribution includes a connector for HDFS, as well as an S3a connector for the Amazon S3 API and a Swift connector for the OpenStack Swift API.

A task writes output to storage through the Hadoop *FileOutputCommitter*. Since each task execution attempt needs to write an output file of the same name, Hadoop employs a rename strategy, where each execution attempt writes its own task temporary file. At *task commit*, the output committer renames the task temporary file to a job temporary file. Task commit is done by the executors, so it occurs in parallel. And then when all of the tasks of a job complete, the driver calls the output committer to do *job commit*, which renames the job temporary files to their final names. Job commit occurs in the driver after all of

the tasks have committed and does not benefit from parallelism. This two stage strategy of task commit and then job commit ensures fault tolerance, i.e., that the output contains just a single complete output file for each task despite multiple executions due to failures and speculation.

Hadoop also writes a zero length object with the name _SUCCESS when a job completes successfully, so the case of incomplete results can easily by identified by the absence of a _SUCCESS object. This enables a new version of the file output committer algorithm (called version 2), where the task temporary files are renamed to their final names at task commit and job commit is largely reduced to the writing of the _SUCCESS object. However, as of Hadoop 2.7.3, this algorithm is not yet the default output committer.

Hadoop is highly distributed and thus it keeps its state in its storage system, e.g., HDFS or object storage. In particular, the output committer determines what temporary objects need to be renamed through "directory" listings, i.e., it lists the "directory" of the output dataset to find the "directory" and files holding task temporary and job temporary output. In object stores this is done through container listing operations. However, due to eventual consistency a container listing may not contain an object that was just successfully created, or it may still contain an object that was just successfully deleted. This can lead to situations where some of the legitimate output objects do not get renamed by the output committer, so that the output of the Spark/Hadoop job will be incomplete.

This situation occurs when speculation is enabled, and thus, despite the benefits of speculation, Spark users are encouraged to run with it disabled. Furthermore, in order to avoid the dangers of eventual consistency entirely, Spark users are often encouraged to copy their input data to HDFS, run their Spark job over the data in HDFS, and then when it is complete, copy the output from HDFS back to object storage. Note, however, that this adds considerable overhead. Existing solutions to this problem require a consistent storage system in addition to object storage[14][15][16].

Netflix and Amazon have implemented solutions to eventual consistency through S3mper and EMR, respectively. These solutions employ a second consistent storage system, Amazon dynamo, in addition to object storage. The Hadoop open source community is working on a similar solution called S3Guard for its S3a storage connector.

6.2 Motivation

To motivate the need for Stocator we describe the sequence of interactions between Spark and its storage system for a program that executes a single task that produces a single output object as shown in fig. 6.2.

At the beginning of a job, the Spark driver and executor recursively create the directories for the task temporary, job temporary and final output. Then, the task outputs the task temporary file. At task commit the executor lists the task temporary directory, and renames the file it finds to its job temporary

[14] http://techblog.netflix.com/2014/01/s3mper-consistency-in-cloud.html
[15] https://aws.amazon.com/blogs/aws/emr-consistent-file-system/
[16] https://issues.apache.org/jira/browse/HADOOP-13345

```
val data = Array(1)
val distData = sc.parallelize(data)
val finalData = distData.coalesce(1)
finalData.saveAsTextFile("hdfs://res/data.txt")
```

Fig. 6.2 A Spark program that executes a single task that produces a single output object.

Table 6.1 Breakdown of REST operations by type for the Spark program that creates an output consisting of a single object.

	HEAD Object	PUT Object	COPY Object	DELETE Object	GET Cont.	Total
Hadoop-Swift	25	7	3	8	5	48
S3a	71	5	2	4	35	117
Stocator	4	3	–	–	1	8

name. At job commit the driver recursively lists the job temporary directories and renames the file it finds to its final names. Finally, the driver writes the _SUCCESS object.

When this same Spark program runs with the Hadoop Swift or S3a connectors, these file operations are translated to equivalent operations on objects in the object store. These connectors use PUT to create zero byte objects representing the directories, after first using HEAD to check if objects for the directories already exist. When listing the contents of a directory, these connectors descend the "directory tree" listing each directory. To rename objects these connectors use PUT or COPY to copy the object to its new name and then use DELETE on the object at the old name. All of the zero byte directory objects also need to be deleted. Overall the Hadoop Swift connector executes 48 REST operations and the S3a connector executes 117 operations. Table 6.1 shows the breakdown according to operation type.

In the next section we describe Stocator, which leverages object storage semantics to replace the temporary file/rename paradigm and takes advantage of hierarchal naming to avoid the creation of "directory" objects. For the Spark program in fig. 6.2 Stocator executes just 8 REST operations: 3 PUT object, 4 HEAD object and 1 GET container.

6.3 Stocator Logic

The right side of fig. 6.1 shows how Stocator fits underneath HMRCC; it implements the Hadoop Filesystem Interface just like the other storage connectors. Below we describe the basic Stocator protocol; and then how it streams data, deals with eventual consistency, and reduces operations on the read path. Finally we provide several examples of the protocol in action.

6.3.1 Basic Stocator protocol

The overall strategy used by Stocator to avoid rename is to write output objects directly to their final name and then to determine which objects actually belong to the output at the time that the output is read by its consumer, e.g., the next Spark job in a sequence of jobs. Stocator does this in a way that preserves the fault tolerance model of Spark/Hadoop and enables speculation. Below we describe the components of this strategy.

As described in section 6.1 the driver orchestrates the execution of a Spark application. In particular, the driver is responsible for creating a "directory" to hold an application's output dataset. Stocator uses this "directory" as a marker to indicate that it wrote the output. In particular, Stocator writes a zero byte object with the name of the dataset and object metadata that indicates that the object was written by Stocator. All of the dataset's parts are stored hierarchically under this name.

Then when a Spark task asks to create a temporary object for its part through HMRCC, Stocator recognizes the pattern of the name and writes the object directly to its final name so it will not need to be renamed. If Spark executes a task multiple times due to failures, slow execution or speculative execution, each execution attempt is assigned a number. The Stocator object naming scheme includes this attempt number so that individual attempts can be distinguished.

Finally, when all tasks have completed successfully, Spark writes a _SUCCESS object through HMRCC; the presence of a _SUCCESS object means that there was a correct execution for each task and that there is an object for each part in the output. Notice that by avoiding rename, Stocator also avoids the need for list operations during task and job commit that may lead to incorrect results due to eventual consistency.

6.3.2 Alternatives for reading an input dataset

Stocator delays the determination of which parts belong to an output dataset until it reads the dataset as input. We consider two options.

The first option is simpler to implement since it can be done entirely in the implementation of Stocator. It depends on the assumption that Spark exhibits fail-stop behavior, i.e., that a Spark server executes correctly until it halts. After determining that the dataset was produced by Stocator through reading the metadata from the object written with the dataset's name, and checking that the _SUCCESS object exists, Stocator lists the object parts belonging to the dataset through a GET container operation. If there are objects in the list representing multiple execution attempts for same task, Stocator will choose the largest. Given the fail-stop assumption, the fact that all successful execution attempts write the same output, and that it is certain that at least one attempt succeeded (otherwise there would not be a _SUCCESS object), this is the correct choice.

At the completion of a Spark job, the second option includes the creation of a manifest inside the _SUCCESS object that contains a list of all the successful task execution attempts completed by the job. Now after determining that the dataset was produced by Stocator through reading the metadata from the object written with the dataset's name, and checking that the _SUCCESS object exists, Stocator

reads the manifest of successful task execution attempts from the _SUCCESS object. Stocator uses the manifest to reconstruct the list of constituent object parts of the dataset. In particular, the construction of the object part names follows the same pattern used when the parts were written.

The benefit of the second option is that it solves the remaining eventual consistency issue by constructing the object names from the manifest rather than issuing a REST command to list the object parts, which may not return a correct result in the presence of eventual consistency. However, given that our primary target for Stocator is IBM Cloud Object Storage and that its container listing is immediately consistent with respect to the writing and deleting of objects, we have not had the need to implement this option.

6.3.3 Streaming of output

When Stocator outputs data it streams the data to the object store as the data is produced using chunked transfer encoding. Normally the total length of the object is one of the parameters of a PUT operation and thus needs to be known before starting the operation. Since Spark produces the data for an object on the fly and the final length of the data is not known until all of its data is produced, this would mean that Spark would need to store the entire object data prior to starting the PUT. To avoid running out of memory, a storage connector for Spark can store the object in the Spark server's local file system as the connector produces the object's content, and then read the object back from the file to do the PUT operation on the object store. Indeed this is what the default Hadoop Swift and S3a connectors do. Instead Stocator leverages HTTP chunked transfer encoding, which is supported by the Swift API. In chunked transfer encoding the object data is sent in chunks, the sender needs to know the length of each chunk, but it does not need to know the final length of the object content before starting the PUT operation. S3a has an optional feature, not activated by default, called fast upload, where it leverages the multi-part upload feature of the S3 API. This achieves a similar effect to chunked transfer encoding except that it uses more memory since the minimum part size for multi-part upload is larger than for chunked transfer.

6.3.4 Optimizing the read path

We describe several optimizations that Stocator uses to reduce the number of operations on the read path.

The first optimization can remove a HEAD operation that occurs just before a GET operation for the same object. In particular, the storage connector often reads the metadata of an object just before its data. Typically this is to check that the object exists and to obtain the size of the object. In file systems this is performed by two different operations. Accordingly a naive implementation for object storage would read object metadata through a HEAD operation, and then read the data of the object itself through a GET operation. However, object store GET operations also return the metadata of an object together with its data. In many of these cases Stocator is able to remove the HEAD operation, which can greatly reduce the overall number of operations invoked on the underlying object storage system.

Table 6.2 Possible operations performed by the Spark application showed in fig. 6.3

	Hadoop Map Reduce Client Core	Stocator
1	PUT /res/data.txt/_temporary/0/_temporary/attempt_201512062056_0000_m_000000_0/part-00000	PUT /res/data.txt/part-00000_attempt_201512062056_0000_m_000000_0
2	PUT /res/data.txt/_temporary/0/_temporary/attempt_201512062056_0000_m_000000_0/part-00001	PUT /res/data.txt/part-00001_attempt_201512062056_0000_m_000000_0
3	PUT /res/data.txt/_temporary/0/_temporary/attempt_201512062056_0000_m_000000_0/part-00002	PUT /res/data.txt/part-00002_attempt_201512062056_0000_m_000000_0
4	PUT /res/data.txt/_temporary/0/_temporary/attempt_201512062056_0000_m_000000_1/part-00002	PUT /res/data.txt/part-00002_attempt_201512062056_0000_m_000000_1
5	PUT /res/data.txt/_temporary/0/_temporary/attempt_201512062056_0000_m_000000_2/part-00002	PUT /res/data.txt/part-00002_attempt_201512062056_0000_m_000000_2
6	DELETE /res/data.txt/_temporary/0/_temporary/attempt_201512062056_0000_m_000000_0/part-00002	DELETE /res/data.txt/part-00002_attempt_201512062056_0000_m_000000_0
7	DELETE /res/data.txt/_temporary/0/_temporary/attempt_201512062056_0000_m_000000_2/part-00002	DELETE /res/data.txt/part-00002_attempt_201512062056_0000_m_000000_2
8	Task commits and job commit generate 2 pairs of COPY and DELETE for each successful attempt	No operations are performed here
9	PUT /res/data.txt/_SUCCESS	PUT /res/data.txt/_SUCCESS

```
val data = Array(1, 2, 3)
val distData = sc.parallelize(data)
distData.saveAsTextFile("swift2d://res.sl/data.txt")
```

Fig. 6.3 A Spark program where three tasks each write an object part.

A second optimization is caching the results of HEAD operations. A basic assumption of Spark is that the input is immutable. Thus, if a HEAD is called on the same input object multiple times, it should return the same result. Stocator uses a small cache to reduce these calls.

6.3.5 Examples

We show here some examples of Stocator at work. For simplicity we focus on Stocator's interaction with HMRCC to eliminate the rename paradigm and so we do not show all of the requests that HMRCC makes on Stocator, e.g., to create/delete "directories" and check their status.

Figure 6.3 shows a simple Spark program that will be executed by three tasks, each task writing its part to the output dataset called *data.txt* in a container called *res*. The *swift2d:* prefix in the URI for the output dataset indicates that Stocator is to be used as the storage connector. Table 6.2 shows the operations that can be executed by our example in different situations.

Lines 1-3 and 8-9 are executed when each task runs exactly once and the program completes successfully. We show the requests that HMRCC generates; for each task it issues one request to create a temporary object and two requests to "rename" it (copy to a new name and delete the object at the former name). We see that Stocator intercepts the pattern for the temporary name that it receives from HMRCC, and creates the final names for the objects directly. At the end of the run Spark creates the _SUCCESS object.

Lines 1-5, instead, shows an execution where Spark decides to execute Task 2 three times, i.e., three attempts. This could be because the first and second attempts failed or due to speculation because they were slow. Notice that Stocator includes the attempt number as part of the name of the objects that it creates.

By adding lines 6-9 to the previous, we show what happens when Spark is able to clean up the results from the duplicate attempts to execute Task 2. In particular, Spark aborts attempts 0 and 2, and commits attempt 1. When Spark aborts attempts 0 and 2, HMRCC deletes their corresponding temporary objects. Stocator recognizes the pattern for the temporary objects and deletes the corresponding objects that it created.

If Spark is not able to clean up the results from the duplicate attempts to execute Task 2, we have lines 1-5 and 8-9. In particular, we see that Stocator created five object parts, one each for Tasks 0 and 1, and three for Task 2 due to its extra attempts. We assume as in the previous situation that it is attempt 1 for Task 2 that succeeded. Stocator recognizes this through the manifest stored in the _SUCCESS object.

6.4 Methodology

We describe the experimental platform, deployment scenarios, workloads and performance metrics that we use to evaluate Stocator.

6.4.1 Experimental Platform

Our experimental infrastructure includes a Spark cluster, an IBM Cloud Object Storage (formerly Cleversafe) cluster, Keystone, and Graphite/Grafana. The Spark cluster consists of three bare metal servers. Each server has a dual Intel Xeon E52690 processor with 12 hyper-threaded 2.60 GHz cores (so 48 hyper-threaded cores per server), 256 GB memory, a 10 Gbps NIC and a 1 TB SATA disk. That means that the total parallelism of the Spark cluster is 144. We run 12 executors on each server; each executor gets 4 cores and 16 GB of memory. We use Spark submit to run the workloads and the driver runs on one of the Spark servers (always the same server). We use the standalone Spark cluster manager.

Our IBM Cloud Object Storage (COS)[17] cluster also runs on bare metal. It consists of two Accessers, front end servers that receive the REST commands and then orchestrate their execution across twelve Slicestors, which hold the storage. Each Accesser has two 10 Gbps NICs bonded to yield 20 Gbps. Each Slicestor has twelve 1 TB SATA disks for data. The Information Dispersal Algorithm (IDA) or erasure code is (12, 8, 10), which means that the erasure code splits the data into 12 parts, 8 parts are needed to read the data, and at least 10 parts need to be written for a write to complete. IBM COS exposes multiple object APIs; we use the Swift and S3 APIs.

We employ HAProxy for load balancing. It is installed on each of the Spark servers and configured in round-robin so that connections opened by a Spark server with the object storage alternate between Accessers. Each of the three Spark servers has a 10 Gbps NIC thus, the maximum network bandwidth between the Spark cluster and the COS cluster is 30 Gbps.

Keystone and Graphite/Grafana run on virtual machines. Keystone provides authentication/authorization for the Swift API. We collect monitoring data on Graphite and view it through Grafana to check that there are no unexpected bottlenecks during the performance runs. In particular we use the Spark monitoring interface and the collectd daemon to collect monitoring data from the Spark servers, and we use the Device API of IBM COS to collect monitoring data from the Accessers and the Slicestors.

6.4.2 Deployment scenarios

In our experiments, we compare Stocator with the Hadoop Swift and S3a connectors. By using different configurations of these two connectors, we define six scenarios: (i) Hadoop-Swift Base (**H-S Base**), (ii) S3a Base (**S3a Base**), (iii) Stocator Base (**Stocator**), (iv) Hadoop-Swift Commit V2 (**H-S Cv2**), (v) S3a Commit V2 (**S3a Cv2**) and (vi) S3a Commit V2 + Fast Upload (**S3a Cv2+FU**). These scenarios are split into 3 groups according to the optional optimization features that are active. The first group, with the suffix *Base*, uses connectors out of the box, meaning that no optional features are active. The

[17] https://www.ibm.com/cloud-computing/products/storage/object-storage/cloud/

second group, with the suffix *Commit V2*, uses the version 2 of Hadoop FileOutputCommitter that reduces the number of copy operations on the object storage. The last group, with the suffix *Commit V2 + Fast Upload*, uses both version 2 of Hadoop FileOutputCommitter and an optimization feature of S3a called S3AFastOutputStream that streams data to the object storage as it is produced (as described in Section 6.3). We decided to compare Stocator to the **Base** scenarios, because the optional features are experimental and not always stable.

All experiments run on *Spark 2.0.1* with a patched[18] version of Hadoop 2.7.3. This patch allows us to use, for the S3a scenarios, *Amazon SDK version 1.11.53* instead of version 1.7.4. The Hadoop-Swift scenarios run with the default Hadoop-Swift connector that comes with Hadoop 2.7.3. Finally, the Stocator scenario runs with *stocator 1.0.8*.[19]

6.4.3 Benchmark and Workloads

To study the performance of our solution we use several workloads from popular benchmark suites that cover different kinds of applications. The workloads span from simple applications that target a single and specific feature of the connectors (micro benchmarks), to complex applications composed by several jobs (macro benchmarks).

The micro benchmarks include three applications: (i) Read-only, (ii) Write-only and (iii) Copy. The Read-only application reads two different text datasets, one whose size is 46.5 GB and the second 465.6 GB, and counts the number of lines in them. For the Write-only application we use the popular Teragen application, available in the Spark example suite, that only performs write operations, creating a dataset of 46.5 GB. The last application that we use for our micro benchmark set is what we call the Copy application; it copies the small dataset used by the Read-only application.

We also use three macro benchmarks. The first, Wordcount from Intel Hi-Bench [Huang et al., 2010] test suite, is the "Hello World" application for parallel computing. It is a read-intensive workload, that reads an input 46.5 GB text file, computes the number of times each word occurs in the file and then writes a much smaller output file (1.3 MB) containing the word counts. The second macro benchmark, Terasort, is a popular application used to understand the performance of large scale computing frameworks like Spark and Hadoop. Its input dataset is the output of the Teragen application used in the micro benchmarks. The third macro benchmark,TPC-DS, is the Transaction Processing Performance Council's decision-support benchmark test[Nambiar and Poess, 2006] implemented with DataBricks' Spark-Sql-Perf library[20]. It executes several complex queries on files stored in Parquet format[21]; the input dataset size is 50 GB, which is compressed to 13.8 GB when converted to Parquet. The query set that we use to perform our experiments is composed of the following 8 TPC-DS queries: q34, q43, q46, q59, q68, q73, q79 and ss_max. These are the queries from the *Impala* subset that work with the Hadoop-Swift connector. Stocator and S3a support all of the queries in the Impala subset.

[18] https://issues.apache.org/jira/browse/HADOOP-12269
[19] These were the latest official releases of these software components at the time of writing this chapter.
[20] https://github.com/databricks/spark-sql-perf
[21] https://parquet.apache.org/

The inputs and outputs for the Read-only, Copy, Wordcount, Teragen and Terasort benchmarks are divided into 128 MB objects. We also run Spark with a partition size of 128 MB.

6.4.4 Performance metrics

We evaluate the different connectors and scenarios by using metrics that target the various optimization features. As a general metric we use the total runtime of the application; this provides a quick overview of the performance of a specific scenario. To delve into the reason behind the performance we use two additional metrics. The first is the number of REST calls – and their type; with this metric we are able to understand the load on the object storage imposed by the connector. The second metric is the number of bytes read from, written to and copied in the object storage; this also help us to understand the load on the object storage imposed by the connectors.

6.5 Experimental Evaluation

We now present a comparative analysis between the different scenarios that we defined in section 6.4.2. We first show the benefit of Stocator through the average run time of the different workloads. Then we compare the number of REST operations issued by the Compute Layer toward the Object Storage and the relative cost for these operations charged by cloud object store services. Finally we compare the number of bytes transferred between the Compute Layer and the Object Storage.

6.5.1 Reduction in run time

For each workload we ran each scenario ten times. Table 6.3 shows the speedups that we obtain when using Stocator with respect to the other connectors. We see a relationship between Stocator performance and the workload; the more write operations performed, the greater the benefit obtained. On the one hand the write-only workloads, like Teragen, run 18 time faster with Stocator compared to the other out of the box connectors, 4 time faster when we enable FileOutputCommitter Version 2, and 1.5 times faster when we also add the S3AFastOutputStream feature. On the other hand, workloads more skewed toward read operations, like Wordcount, have lower speedups.

These results are possible thanks to the algorithm implemented in Stocator. Unlike the alternatives, Stocator removes the rename – and thus copy – operations completely. In contrast, the other connectors, even with FileOutputCommitter Version 2, must still rename each output object once, although the overhead of the remaining renames is partially masked since they are carried out by the executors in parallel.

Stocator performs slightly worse than S3a on two of the workloads that contain only read operations (no writes), Read-only 50 GB and TPC-DS, and virtually the same for the larger 500 GB Read-only workload. We have identified a small start-up cost that we have not yet removed from Stocator that can explain the difference between the results for the 50 GB and 500 GB Read-only workload.[22] As

[22]Since writing this chapter we have removed start-up costs and improved the performance of the read-path of Stocator.

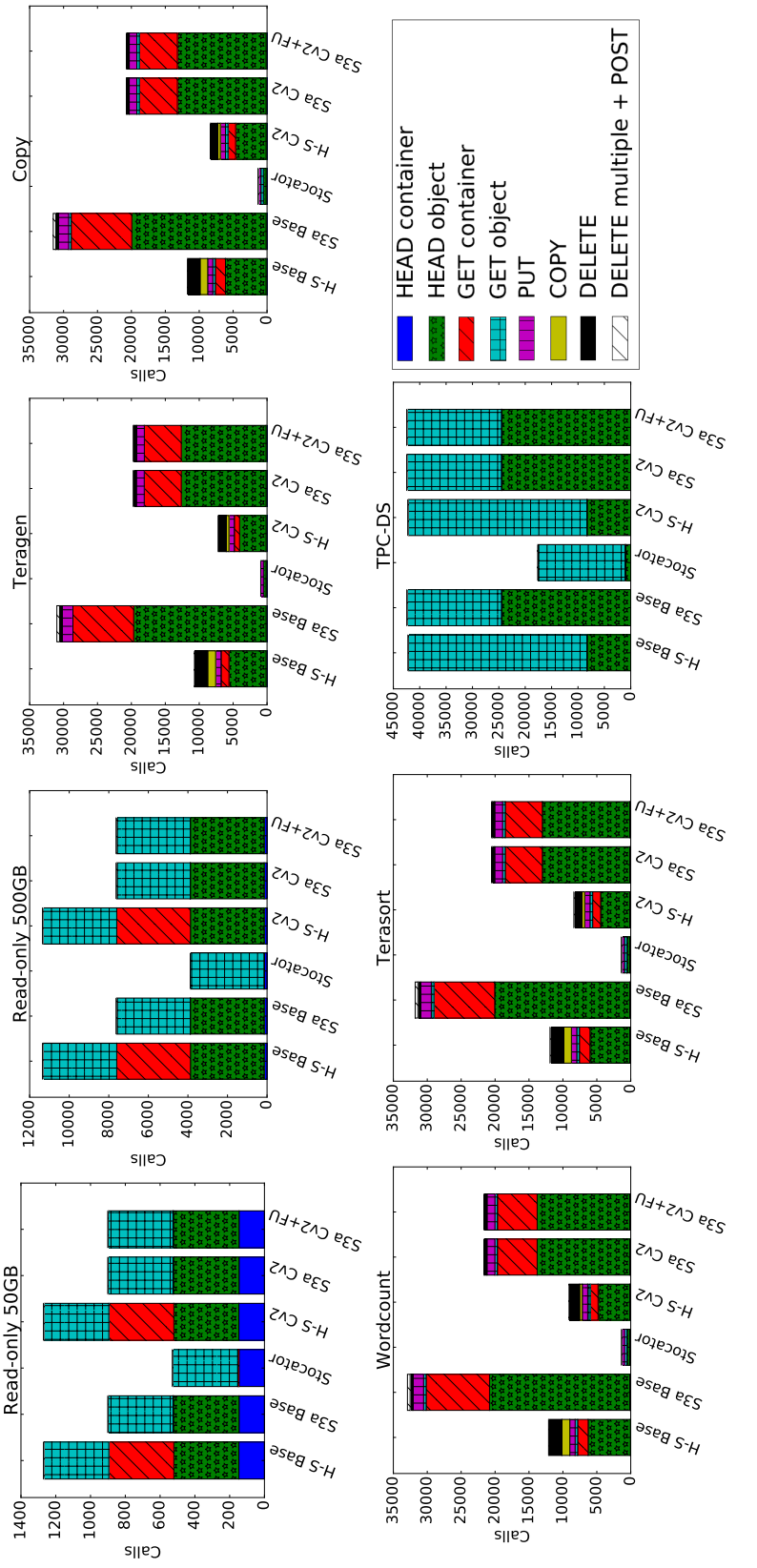


Fig. 6.4 Benchmarks REST calls comparison

Table 6.3 Workload speedups when using Stocator

	Read-Only 50GB	Read-Only 500GB	Teragen	Copy	Wordcount	Terasort	TPC-DS
Hadoop-Swift Base	x1.09	x1.55	x16.09	x9.12	x2.29	x8.10	x0.91
S3a Base	x0.96	x1.00	x18.03	x10.33	x1.82	x8.86	x0.94
Stocator	x1	x1	x1	x1	x1	x1	x1
Hadoop-Swift Cv2	x1.07	x1.55	x4.41	x2.57	x1.57	x2.64	x0.92
S3a Cv2	x1.02	x1.00	x4.37	x2.72	x1.05	x2.64	x0.93
S3a Cv2 + FU	x1.02	x1.00	x1.46	x1.27	x1.05	x1.25	x0.93

Table 6.4 Ratio of REST calls compared to Stocator

	Read-Only 50GB	Read-Only 500GB	Teragen	Copy	Wordcount	Terasort	TPC-DS
Hadoop-Swift Base	x2.41	x2.92	x11.51	x9.18	x9.21	x8.94	x2.39
S3a Base	x1.71	x1.96	x33.74	x24.93	x25.35	x24.23	x2.40
Stocator	x1	x1	x1	x1	x1	x1	x1
Hadoop-Swift Cv2	x2.41	x2.92	x7.72	x6.55	x6.92	x6.29	x2.39
S3a Cv2	x1.71	x1.96	x21.15	x16.18	x16.44	x15.41	x2.40
S3a Cv2 + FU	x1.71	x1.96	x21.15	x16.18	x16.44	x15.41	x2.40

Table 6.5 Financial cost for REST calls compared to Stocator for IBM, AWS, Google and Azure infrastructure

	Read-Only 50GB	Read-Only 500GB	Teragen	Copy	Wordcount	Terasort	TPC-DS
Hadoop-Swift Base	x9.72	x13.67	x8.23	x8.60	x8.58	x8.57	x2.23
S3a Base	x1.63	x1.94	x27.82	x26.74	x26.84	x25.88	x2.25
Stocator	x1	x1	x1	1	x1	x1	x1
Hadoop-Swift Cv2	x9.72	x13.67	x5.24	x5.86	x5.85	x5.81	x2.23
S3a Cv2	x1.63	x1.94	x17.59	x17.29	x17.36	x16.40	x2.25
S3a Cv2 + FU	x1.63	x1.94	x17.55	x17.29	x17.34	x16.40	x2.25

expected the results for the read-only workloads for S3a and Hadoop-Swift connectors are virtually the same with and without the FileOutputCommitter Version 2 and S3AFastOutputStream features; these features optimize the write path and do not affect the read path.

6.5.2 Reduction in the number of REST calls

Next we look at the number of REST operations executed by Spark in order to understand the load generated on the object storage infrastructure. Figure 6.4 shows that, in all the workloads, the scenario that uses Stocator achieves the lowest number of REST calls and thus the lowest load on the object storage.

When looking at Read-only with both 50 and 500 GB dataset, the scenario with Hadoop-Swift has the highest number of REST calls and more than double compared to the scenario with Stocator. The Hadoop-Swift connector does many more GET calls on containers to list their contents. Compared to S3a, Stocator is optimized to reduce the number of HEAD calls on the objects. We see this consistently for all of the workloads.

In write-intensive workloads, Teragen and Copy, we see that the scenarios that use S3a as the connector have the highest number of REST calls while Stocator still has the lowest. Compared to Hadoop-Swift and Stocator, S3a performs many more HEAD calls for the objects and GET for the containers. Stocator also does not need to create temporary "directories" objects, thus uses far fewer HEAD requests, and does not need to DELETE objects; this is possible because our algorithm is conceived to avoid renaming objects after a task or job completes. Table 6.4 shows the number of REST calls that is possible to save by using Stocator. We observe that, for write-intensive workloads, Stocator issues 6 to 11 times less REST calls compared to Hadoop-Swift and 15 to 33 times less compared to S3a, depending on the optimization features.

Having a low load on the Object Storage has advantages both for the data scientist and the storage providers. On the one hand, cloud providers will be able to serve a bigger pool of consumers and give them a better experience. On the other hand, since most public providers charge fees based on the number of operations performed on the storage tier, reducing the operations results in a lower cost for the data scientists. Table 6.5 shows the relative costs for the REST operations. For the workloads with write (Teragen, Copy, Terasort and Wordcount) Stocator is 16 to 18 times less expensive than S3a run with FileOutputCommitter version 2, and 5 to 6 times less expensive than Hadoop-Swift. To calculate the cost ratio we used the pricing models of IBM[23], AWS[24], Google[25] and Azure[26]; given that the models are very similar we report the average price.

As an additional way of measuring the load on the object storage and confirming the fact that Stocator does not perform COPY (or DELETE) operations we present the number of bytes read and written to the object storage. From fig. 6.5 we see that Stocator does not write more data than needed on the storage. In

[23] http://www-03.ibm.com/software/products/en/object-storage-public/#othertab2
[24] https://aws.amazon.com/s3/pricing/
[25] https://cloud.google.com/storage/pricing
[26] https://azure.microsoft.com/en-us/pricing/details/storage/blobs/

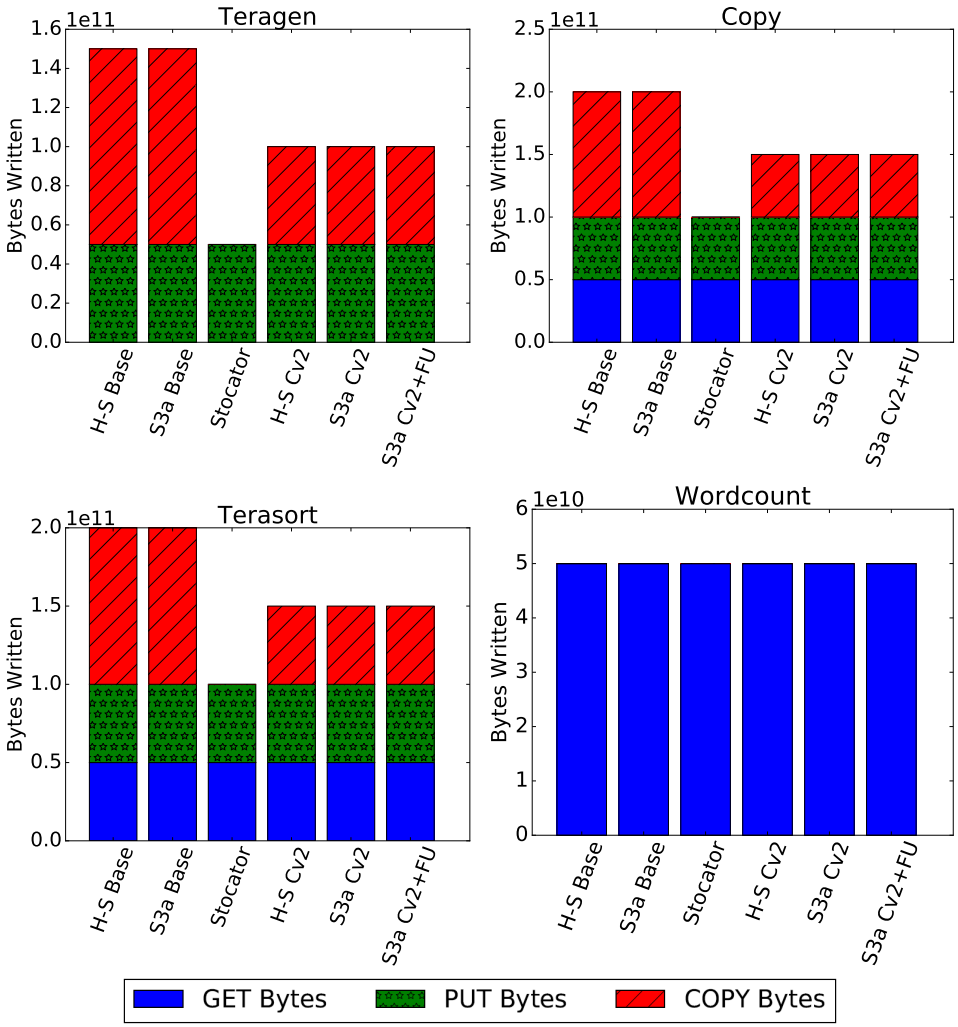


Fig. 6.5 Object Storage bytes read/written comparison

contrast we confirm that Hadoop-Swift and S3a base write each object three times – one from the PUT and two from the COPY – while Stocator only does it once. Only by enabling FileOutputCommitter Version 2 in Hadoop, it is possible to reduce the COPY operations to one, but this is still one more object copy compared to Stocator. We show only the workloads that have write operations since during a read-only workload, the number of bytes read from the object storage are identical for all of the connectors and scenarios (as we see from the Wordcount workload in fig. 6.5 where the number of bytes written is very small). As expected the S3a scenario that uses the S3AFastOutputStream optimization gains no benefit with respect to the number of bytes written to the object storage.

6.6 Summary

We have presented a high performance object storage connector for Apache Spark called Stocator, which has been made available to the open source community[27]. Stocator overcomes the impedance mismatch of previous open source connectors with their storage, by leveraging object storage semantics rather than trying to treat object storage as a file system. In particular Stocator eliminates the rename paradigm without sacrificing fault tolerance or speculative execution. It also deals correctly with the eventually consistent semantics of object stores without the need to use an additional consistent storage system. Finally, Stocator leverages HTTP chunked transfer encoding to stream data as it is produced to object storage, thereby avoiding the need to first write output to local storage.

We have compared Stocator's performance with the Hadoop Swift and S3a connectors over a range of workloads and found that it executes far less operations on object storage, in some cases as little as one thirtieth. This reduces the load both for client software and the object storage service, as well as reducing costs for the client. Stocator also substantially increases the performance of Spark workloads, especially write intensive workloads, where it is as much as 18 times faster than alternatives.

[27] https://github.com/SparkTC/stocator

Chapter 7

Conclusions and Perspectives

In this thesis we have presented contributions towards the improving of data-center efficiency in term of system responsiveness. Solutions have been offered to achieve different goals:

- **Raise the level of Abstraction.** We defined, for the first time, a high-level construct to represent analytic applications, focusing on their heterogeneity, and their end-to-end life-cycle. We established a new scheduling problem, and proposed a flexible heuristic capable of handling heterogeneous requests, as well as a variety of scheduling policies, with the ultimate objective of improving system responsiveness under heavy loads. We evaluated our scheduling policy using realistic, large-scale workload traces and show it consistently outperforms the baseline approach. Finally, we built a full-fledge system which materializes the ideas of analytic applications and their scheduling. Using our new heuristic, we were able to achieve substantial improvements in terms of system responsiveness and cluster allocation.

- **Cluster Utilization.** We presented the system design for a dynamic resource allocation mechanism, which can be generally applied to existing cluster management frameworks. We targeted a specific family of analytic application schedulers, and materialized our ideas for such schedulers. We introduced a novel application of state-of-the-art machine learning methodologies for accurate forecasting of resource utilization, featuring a probabilistic treatment that allows quantification of uncertainty. Confidence information was used to steer system parameters to safeguard against unexpected resource demand peaks. We performed an extensive simulation campaign using publicly available production traces from Google data-centers. We compared our approach to that of Borg, and discuss about the trade-off that an optimistic vs. a pessimistic approach to application preemption entails. Finally, we presented the design of a prototype implementation of our system, that we use in an academic cluster serving students and researchers. Our results indicated substantial improvements in terms of efficiency, which translate in a system capable of ingesting a heavier workload with the same number of machines.

- **Compute and Storage Disaggregation.** We performed an extensive measurement campaign on a private cloud computing environment involving the combination of several analytics services. For

each deployment scenario, we investigated the performance of a variety of application workloads, including read/write intensive, business intelligence and machine learning applications. We presented an intuitive notion of data locality that can be used as a proxy to rank different service compositions, in terms of expected performance. We critically examined the validity of our intuition as a function of application workloads, and identify and explain outliers. We showed experimental evidence of the impedance mismatch between large-scale computing framework and two important storage layers – object stores and elastic block stores – and deduced mechanism to mitigate negative effects on performance.

- **Object Storage and large-scale computing frameworks impedance mismatch.** We presented the design of a novel storage connector for Hadoop and Spark that leverages object storage semantics to provide high performance and correct execution in the face of faults and speculation. We proved that this solution works correctly despite the eventually consistent semantics of object storage, yet without requiring additional strongly consistent storage.

With the work presented in this dissertation we reduced the gap between resource allocation and utilization. More work can be done in order to study new techniques that can better model the resources utilization inside a cluster; in this way we can lower even further the number of failures due to resources contention and reduce the resources slack. In our prose and experiments we always considered just CPU and Memory as main resources, however other type of resources can be taken into account, like Network bandwidth and disk I/O. Additionally, our dynamic solution is in a prototype stage. Public providers must guarantee some Service Level Objective (SLO) and Service Level Agreement (SLA) to their user. More work must be done to understand the impact of our pessimistic approach on such service level constraints. Finally, the machine learning methodology that we proposed, may fail with interactive applications due to the their “human-in-the-loop” nature. It is challenging to predict when an user is going to interact with an application that has been deemed idle by just looking at the resource utilization. At the same time, it is not an easy task to understand when an interactive application is in an idle state; some resources (e.g., Memory) can be kept busy even when applications are not producing any work.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M. et al. [2016], Tensorflow: A system for large-scale machine learning., *in* 'OSDI', Vol. 16, pp. 265–283.

Adhikari, R. and Agrawal, R. K. [2013], 'An Introductory Study on Time Series Modeling and Forecasting', *ArXiv e-prints* .

Alipourfard, O., Liu, H. H., Chen, J., Venkataraman, S., Yu, M. and Zhang, M. [2017], Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics, *in* 'NSDI', pp. 469–482.

Ananthanarayanan, G., Douglas, C., Ramakrishnan, R., Rao, S. and Stoica, I. [2012], True elasticity in multi-tenant data-intensive compute clusters, *in* 'Proceedings of the Third ACM Symposium on Cloud Computing', ACM, p. 24.

Ananthanarayanan, G., Ghodsi, A., Shenker, S. and Stoica, I. [2011], Disk-locality in datacenter computing considered irrelevant., *in* 'HotOS', Vol. 13, pp. 12–12.

Arnold, J. [2014], *Openstack swift: Using, administering, and developing for swift object storage*, " O'Reilly Media, Inc.".

Babaioff, M., Mansour, Y., Nisan, N., Noti, G., Curino, C., Ganapathy, N., Menache, I., Reingold, O., Tennenholtz, M. and Timnat, E. [2017], Era: A framework for economic resource allocation for the cloud, *in* 'Proceedings of the 26th International Conference on World Wide Web Companion', International World Wide Web Conferences Steering Committee, pp. 635–642.

Boutin, E., Ekanayake, J., Lin, W., Shi, B., Zhou, J., Qian, Z., Wu, M. and Zhou, L. [2014], Apollo: Scalable and coordinated scheduling for cloud-scale computing., *in* 'OSDI', Vol. 14, pp. 285–300.

Box, G., Jenkins, G. and Reinsel, G. [2008], *Time Series Analysis, Forecasting and Control*, Wiley Series in Probability and Statistics, Wiley.

Brockwell, P. J. and Davis, . R. A. [2002], *Introduction to Time Series and Forecasting, Second Edition*, Springer.

Brockwell, P. J. and Davis, R. A. [2016], *Introduction to time series and forecasting*, springer.

Burns, B., Grant, B., Oppenheimer, D., Brewer, E. and Wilkes, J. [2016], 'Borg, omega, and kubernetes', *Queue* **14**(1), 10.

Chalupka, K., Williams, C. K. I. and Murray, I. [2013], 'A framework for evaluating approximation methods for gaussian process regression', *J. Mach. Learn. Res.* **14**(1), 333–350.
URL: *http://dl.acm.org/citation.cfm?id=2502581.2502592*

Curino, C., Difallah, D. E., Douglas, C., Krishnan, S., Ramakrishnan, R. and Rao, S. [2014], Reservation-based scheduling: If you're late don't blame us!, *in* 'Proceedings of the ACM Symposium on Cloud Computing', ACM, pp. 1–14.

Cutajar, K., Bonilla, E., Michiardi, P. and Filippone, M. [2017], Random feature expansions for deep Gaussian processes, *in* 'ICML 2017, 34th International Conference on Machine Learning, 6-11 August 2017, Sydney, Australia', Sydney, AUSTRALIA.
URL: *https://www.eurecom.fr/publication/5214*

Cutajar, K., Bonilla, E. V., Michiardi, P. and Filippone, M. [2016], 'Practical learning of deep gaussian processes via random fourier features', *stat* **1050**, 14.

Dean, J. and Ghemawat, S. [2008], 'Mapreduce: simplified data processing on large clusters', *Communications of the ACM* **51**(1), 107–113.

Delgado, P., Dinu, F., Didona, D. and Zwaenepoel, W. [2016], Eagle: A better hybrid data center scheduler, Technical report, Tech. Rep.

Delgado, P., Dinu, F., Kermarrec, A.-M. and Zwaenepoel, W. [2015], Hawk: Hybrid datacenter scheduling, *in* 'USENIX Annual Technical Conference (USENIX ATC'15)', pp. 499–510.

Delimitrou, C. and Kozyrakis, C. [2013], Paragon: Qos-aware scheduling for heterogeneous datacenters, *in* 'ACM SIGPLAN Notices', Vol. 48, ACM, pp. 77–88.

Delimitrou, C. and Kozyrakis, C. [2014], 'Quasar: resource-efficient and qos-aware cluster management', *ACM SIGPLAN Notices* **49**(4), 127–144.

Delimitrou, C. and Kozyrakis, C. [2016], 'Hcloud: Resource-efficient provisioning in shared cloud systems', *ACM SIGOPS Operating Systems Review* **50**(2), 473–488.

Delimitrou, C., Sanchez, D. and Kozyrakis, C. [2015], Tarcil: reconciling scheduling speed and quality in large shared clusters, *in* 'Proceedings of the Sixth ACM Symposium on Cloud Computing', ACM, pp. 97–110.

Dell'Amico, M., Carra, D. and Michiardi, P. [2016], 'Psbs: Practical size-based scheduling', *IEEE Transactions on Computers* **65**(7), 2199–2212.

Dell'Amico, M., Carra, D., Pastorelli, M. and Michiardi, P. [2014], Revisiting size-based scheduling with estimated job sizes, *in* 'Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2014 IEEE 22nd International Symposium on', IEEE, pp. 411–420.

Dutot, P.-F. et al. [2004], 'Scheduling parallel tasks: Approximation algorithms', *Handbook of scheduling: Algorithms, models, and performance analysis* pp. 26–1.

Frigola-Alcalde, R. [2015], Bayesian Time Series Learning with Gaussian Processes, PhD thesis, University of Cambridge.

Frigola, R., Chen, Y. and Rasmussen, C. E. [2007], Variational Gaussian Process State-Space Models, *in* 'Advances in Neural Information Processing Systems', MIT Press.

Ghit, B. and Epema, D. [2016], Tyrex: Size-based resource allocation in mapreduce frameworks, *in* 'Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on', IEEE, pp. 11–20.

Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S. and Stoica, I. [2011], Dominant resource fairness: Fair allocation of multiple resource types., *in* 'Nsdi', Vol. 11, pp. 24–24.

Goder, A., Spiridonov, A. and Wang, Y. [2015], Bistro: Scheduling data-parallel jobs against live production systems., *in* 'USENIX Annual Technical Conference', pp. 459–471.

Gog, I., Schwarzkopf, M., Gleave, A., Watson, R. N. and Hand, S. [2016], Firmament: Fast, centralized cluster scheduling at scale, Usenix.

Grandl, R., Ananthanarayanan, G., Kandula, S., Rao, S. and Akella, A. [2014], Multi-resource packing for cluster schedulers, *in* 'ACM SIGCOMM Computer Communication Review', Vol. 44, ACM, pp. 455–466.

Grandl, R., Kandula, S., Rao, S., Akella, A. and Kulkarni, J. [2016], G: Packing and dependency-aware scheduling for data-parallel clusters, *in* 'Proceedings of OSDI'16: 12th USENIX Symposium on Operating Systems Design and Implementation', p. 81.

Gu, J., Lee, Y., Zhang, Y., Chowdhury, M. and Shin, K. G. [2017], Efficient memory disaggregation with infiniswap, *in* 'NSDI', pp. 649–667.

Guo, Z., Fox, G. and Zhou, M. [2012], Investigation of data locality and fairness in mapreduce, *in* 'Proceedings of third international workshop on MapReduce and its Applications Date', ACM, pp. 25–32.

Hassan, W. U. and Zwaenepoel, W. [2017], Don't cry over spilled records: Memory elasticity of data-parallel applications and its application to cluster scheduling, *in* 'USENIX Annual Technical Conference (USENIX ATC 17)'.

Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R. H., Shenker, S. and Stoica, I. [2011], Mesos: A platform for fine-grained resource sharing in the data center., *in* 'NSDI', Vol. 11, pp. 22–22.

Huang, S., Huang, J., Dai, J., Xie, T. and Huang, B. [2010], The hibench benchmark suite: Characterization of the mapreduce-based data analysis, *in* 'Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on', IEEE, pp. 41–51.

Hyndman, R. J., Khandakar, Y. et al. [2007], *Automatic time series for forecasting: the forecast package for R*, number 6/07, Monash University, Department of Econometrics and Business Statistics.

Isard, M., Budiu, M., Yu, Y., Birrell, A. and Fetterly, D. [2007], Dryad: distributed data-parallel programs from sequential building blocks, *in* 'ACM SIGOPS operating systems review', Vol. 41, ACM, pp. 59–72.

Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K. and Goldberg, A. [2009], Quincy: fair scheduling for distributed computing clusters, *in* 'Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles', ACM, pp. 261–276.

Karanasos, K., Rao, S., Curino, C., Douglas, C., Chaliparambil, K., Fumarola, G. M., Heddaya, S., Ramakrishnan, R. and Sakalanaga, S. [2015], Mercury: Hybrid centralized and distributed scheduling in large shared clusters., *in* 'USENIX Annual Technical Conference', pp. 485–497.

Kuzmanovska, A., Mak, R. H. and Epema, D. [2014], Dynamically scheduling a component-based framework in clusters, *in* 'Workshop on Job Scheduling Strategies for Parallel Processing', Springer, pp. 129–146.

Kuzmanovska, A., Mak, R. H. and Epema, D. [2016], Koala-f: A resource manager for scheduling frameworks in clusters, *in* 'Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on', IEEE, pp. 80–89.

Leung, J. Y. [2004], *Handbook of scheduling: algorithms, models, and performance analysis*, CRC Press.

Li, H., Ghodsi, A., Zaharia, M., Baldeschwieler, E., Shenker, S. and Stoica, I. [2013], 'Tachyon: Memory throughput i/o for cluster computing frameworks', *memory* **18**, 1.

Lin, X., Meng, Z., Xu, C. and Wang, M. [2012], A practical performance model for hadoop mapreduce, *in* 'Cluster Computing Workshops (CLUSTER WORKSHOPS), 2012 IEEE International Conference on', IEEE, pp. 231–239.

Lo, D., Cheng, L., Govindaraju, R., Ranganathan, P. and Kozyrakis, C. [2015], Heracles: improving resource efficiency at scale, *in* 'ACM SIGARCH Computer Architecture News', Vol. 43, ACM, pp. 450–462.

MacKay, D. J. C. [2003], *Information Theory, Inference & Learning Algorithms*, Cambridge University Press.

Mao, H., Alizadeh, M., Menache, I. and Kandula, S. [2016], Resource management with deep reinforcement learning, *in* 'Proceedings of the 15th ACM Workshop on Hot Topics in Networks', ACM, pp. 50–56.

McHutchon, A. [2015], Nonlinear Modelling and Control using Gaussian Processes, PhD thesis, University of Cambridge.

Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S. et al. [2016], 'Mllib: Machine learning in apache spark', *The Journal of Machine Learning Research* **17**(1), 1235–1241.

Moatti, Y., Rom, E., Gracia-Tinedo, R., Naor, D., Chen, D., Sampe, J., Sanchez-Artigas, M., Garcıa-Lopez, P., Gluszak, F., Deschdt, E. et al. [2017], Too big to eat: Boosting analytics data ingestion from object stores with scoop, *in* 'Data Engineering (ICDE), 2017 IEEE 33rd International Conference on', IEEE, pp. 309–320.

Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P. and Abadi, M. [2013], Naiad: a timely dataflow system, *in* 'Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles', ACM, pp. 439–455.

Nambiar, R. O. and Poess, M. [2006], The making of tpc-ds, *in* 'Proceedings of the 32nd international conference on Very large data bases', VLDB Endowment, pp. 1049–1058.

Nightingale, E. B., Elson, J., Fan, J., Hofmann, O. S., Howell, J. and Suzue, Y. [2012], Flat datacenter storage., *in* 'OSDI', pp. 1–15.

Ousterhout, K., Rasti, R., Ratnasamy, S., Shenker, S., Chun, B.-G. and ICSI, V. [2015], Making sense of performance in data analytics frameworks., *in* 'NSDI', Vol. 15, pp. 293–307.

Ousterhout, K., Wendell, P., Zaharia, M. and Stoica, I. [2013], Sparrow: distributed, low latency scheduling, *in* 'Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles', ACM, pp. 69–84.

Pace, F., Milanesio, M., Venzano, D., Carra, D. and Michiardi, P. [2016], Experimental performance evaluation of cloud-based analytics-as-a-service, *in* 'Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on', IEEE, pp. 196–203.

Pace, F., Venzano, D., Carra, D. and Michiardi, P. [2016], 'Flexible scheduling of distributed analytic applications', *arXiv preprint arXiv:1611.09528* .

Pace, F., Venzano, D., Carra, D. and Michiardi, P. [2017], Flexible scheduling of distributed analytic applications, *in* 'Cluster, Cloud and Grid Computing (CCGRID), 2017 17th IEEE/ACM International Symposium on', IEEE, pp. 100–109.

Padala, P., Shin, K. G., Zhu, X., Uysal, M., Wang, Z., Singhal, S., Merchant, A. and Salem, K. [2007], Adaptive control of virtualized resources in utility computing environments, *in* 'ACM SIGOPS Operating Systems Review', Vol. 41, ACM, pp. 289–302.

Pastorelli, M., Barbuzzi, A., Carra, D., Dell'Amico, M. and Michiardi, P. [2013], Hfsp: size-based scheduling for hadoop, *in* 'Big Data, 2013 IEEE International Conference on', IEEE, pp. 51–59.

Pastorelli, M., Dell'Amico, M. and Michiardi, P. [2014], Os-assisted task preemption for hadoop, *in* 'Distributed Computing Systems Workshops (ICDCSW), 2014 IEEE 34th International Conference on', IEEE, pp. 94–99.

Pruhs, K. et al. [2004], 'Online scheduling', *Handbook of scheduling: algorithms, models, and performance analysis* pp. 15–1.

Quiñonero Candela, J. and Rasmussen, C. E. [2005], 'A unifying view of sparse approximate gaussian process regression', *J. Mach. Learn. Res.* **6**, 1939–1959.

Rabinovici-Cohen, S., Henis, E., Marberg, J. and Nagin, K. [2014], 'Storlet engine: performing computations in cloud storage', *IBM Technical Report H-0320 (August 2014), Tech. Rep.* .

Ragan-Kelley, M., Perez, F., Granger, B., Kluyver, T., Ivanov, P., Frederic, J. and Bussonnier, M. [2014], The jupyter/ipython architecture: a unified view of computational research, from interactive exploration to communication and publication., *in* 'AGU Fall Meeting Abstracts'.

Rahimi, A. and Recht, B. [2007], Random features for large-scale kernel machines, *in* 'NIPS'.

Ranganathan, K. and Foster, I. [2002], Decoupling computation and data scheduling in distributed data-intensive applications, *in* 'High Performance Distributed Computing, 2002. HPDC-11 2002. Proceedings. 11th IEEE International Symposium on', IEEE, pp. 352–358.

Rasley, J., Karanasos, K., Kandula, S., Fonseca, R., Vojnovic, M. and Rao, S. [2016], Efficient queue management for cluster scheduling, *in* 'Proceedings of the Eleventh European Conference on Computer Systems', ACM, p. 36.

Rasmussen, C. E. and Williams, C. K. I. [2006], *Gaussian Processes for Machine Learning*, MIT Press.

Reiss, C., Tumanov, A., Ganger, G. R., Katz, R. H. and Kozuch, M. A. [2012], Heterogeneity and dynamicity of clouds at scale: Google trace analysis, *in* 'Proceedings of the Third ACM Symposium on Cloud Computing', ACM, p. 7.

Reiss, C., Wilkes, J. and Hellerstein, J. L. [2011], Google cluster-usage traces: format + schema, Technical report, Google Inc., Mountain View, CA, USA. Revised 2014-11-17 for version 2.1. Posted at https://github.com/google/cluster-data.

Roman, R.-I., Nicolae, B., Costan, A. and Antoniu, G. [2015], Understanding spark performance in hybrid and multi-site clouds, *in* 'BDAC-15-6th International Workshop on Big Data Analytics: Challenges and Opportunities (in conjunction with SC15)'.

Rupprecht, L., Zhang, R. and Hildebrand, D. [2014], 'Big data analytics on object stores: A performance study', *red* **30**, 35.

Rupprecht, L., Zhang, R., Owen, B., Pietzuch, P. and Hildebrand, D. [2017], Swiftanalytics: Optimizing object storage for big data analytics, *in* 'Cloud Engineering (IC2E), 2017 IEEE International Conference on', IEEE, pp. 245–251.

Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M. and Wilkes, J. [2013], Omega: flexible, scalable schedulers for large compute clusters, *in* 'Proceedings of the 8th ACM European Conference on Computer Systems', ACM, pp. 351–364.

Schwiegelshohn, U. and Yahyapour, R. [1998], Analysis of first-come-first-serve parallel job scheduling, *in* 'SODA', Vol. 98, Citeseer, pp. 629–638.

Seabold, S. and Perktold, J. [2010], Statsmodels: Econometric and statistical modeling with python, *in* '9th Python in Science Conference'.

Sgall, J. [2015], 'Online preemptive scheduling on parallel machines.'.

Shahrad, M., Klein, C., Zheng, L., Chiang, M., Elmroth, E. and Wentzlaf, D. [2017], Incentivizing self-capping to increase cloud utilization, *in* 'ACM Symposium on Cloud Computing 2017 (SoCC'17)', Association for Computing Machinery (ACM).

Shvachko, K., Kuang, H., Radia, S. and Chansler, R. [2010], The hadoop distributed file system, *in* 'Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on', Ieee, pp. 1–10.

Sivasubramanian, S. [2012], Amazon dynamodb: a seamlessly scalable non-relational database service, *in* 'Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data', ACM, pp. 729–730.

Snelson, E. and Ghahramani, Z. [2005], Sparse gaussian processes using pseudo-inputs, *in* 'Proceedings of the 18th International Conference on Neural Information Processing Systems', NIPS, MIT Press, Cambridge, MA, USA, pp. 1257–1264.
URL: *http://dl.acm.org/citation.cfm?id=2976248.2976406*

Sumway, R. H. and Stoffer, D. S. [2006], 'Time series analysis and its applications with r examples', *Time series analysis and its applications with R examples* .

Svensson, A., Solin, A., Särkkä, S. and Schön, T. [2016], Computationally Efficient Bayesian Learning of Gaussian Process State Space Models, *in* 'Proceedings of the 19th International Conference on Artificial Intelligence and Statistics', Vol. 51 of *Proceedings of Machine Learning Research*, PMLR, pp. 213–221.

Tannenbaum, T., Wright, D., Miller, K. and Livny, M. [2001], Condor: a distributed job scheduler, *in* 'Beowulf cluster computing with Linux', MIT press, pp. 307–350.

Thinakaran, P., Gunasekaran, J. R., Sharma, B., Kandemir, M. T. and Das, C. R. [2017], Phoenix: A constraint-aware scheduler for heterogeneous datacenters, *in* 'Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on', IEEE, pp. 977–987.

Trivedi, A., Stuedi, P., Pfefferle, J., Stoica, R., Metzler, B., Koltsidas, I. and Ioannou, N. [2016], 'On the [ir] relevance of network performance for data processing', *Network* **40**, 60.

Turner, R., Deisenroth, M. and Rasmussen, C. [2010], State-Space Inference and Learning with Gaussian Processes, *in* 'Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics', Vol. 9 of *Proceedings of Machine Learning Research*, PMLR, pp. 868–875.

Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S. et al. [2013], Apache hadoop yarn: Yet another resource negotiator, *in* 'Proceedings of the 4th annual Symposium on Cloud Computing', ACM, p. 5.

Venzano, D. and Michiardi, P. [2013], A measurement study of data-intensive network traffic patterns in a private cloud, *in* 'Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing', IEEE Computer Society, pp. 476–481.

Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E. and Wilkes, J. [2015], Large-scale cluster management at google with borg, *in* 'Proceedings of the Tenth European Conference on Computer Systems', ACM, p. 18.

Vernik, G., Factor, M., Kolodner, E. K., Ofer, E., Michiardi, P. and Pace, F. [2017], Stocator: an object store aware connector for apache spark, *in* 'Proceedings of the 2017 Symposium on Cloud Computing', ACM, pp. 653–653.

Vogels, W. [2009], 'Eventually consistent', *Communications of the ACM* **52**(1), 40–44.

Wang, G., Butt, A. R., Pandey, P. and Gupta, K. [2009], A simulation approach to evaluating design decisions in mapreduce setups, *in* 'Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2009. MASCOTS'09. IEEE International Symposium on', IEEE, pp. 1–11.

Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. and Maltzahn, C. [2006], Ceph: A scalable, high-performance distributed file system, *in* 'Proceedings of the 7th symposium on Operating systems design and implementation', USENIX Association, pp. 307–320.

Wierman, A., Harchol-Balter, M. and Osogami, T. [2005], Nearly insensitive bounds on smart scheduling, *in* 'ACM SIGMETRICS Performance Evaluation Review', Vol. 33, ACM, pp. 205–216.

Wilkes, J. [2011], 'More Google cluster data', Google research blog. Posted at http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html.

Xie, J., Yin, S., Ruan, X., Ding, Z., Tian, Y., Majors, J., Manzanares, A. and Qin, X. [2010], Improving mapreduce performance through data placement in heterogeneous hadoop clusters, *in* 'Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on', IEEE, pp. 1–9.

Yan, Y., Gao, Y., Chen, Y., Guo, Z., Chen, B. and Moscibroda, T. [2016], Tr-spark: Transient computing for big data analytics, *in* 'Proceedings of the Seventh ACM Symposium on Cloud Computing', ACM, pp. 484–496.

Yang, X. and Sun, J. [2011], An analytical performance model of mapreduce, *in* 'Cloud Computing and Intelligence Systems (CCIS), 2011 IEEE International Conference on', IEEE, pp. 306–310.

Yang, Y., Kim, G.-W., Song, W. W., Lee, Y., Chung, A., Qian, Z., Cho, B. and Chun, B.-G. [2017], Pado: A data processing engine for harnessing transient resources in datacenters, *in* 'Proceedings of the Twelfth European Conference on Computer Systems', ACM, pp. 575–588.

Yoo, A. B., Jette, M. A. and Grondona, M. [2003], Slurm: Simple linux utility for resource management, *in* 'Workshop on Job Scheduling Strategies for Parallel Processing', Springer, pp. 44–60.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S. and Stoica, I. [2012], Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, *in* 'Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation', USENIX Association, pp. 2–2.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S. and Stoica, I. [2010], 'Spark: Cluster computing with working sets.', *HotCloud* **10**(10-10), 95.

Zhang, C., Ding, C., Ogihara, M., Zhong, Y. and Wu, Y. [2006], A hierarchical model of data locality, *in* 'ACM SIGPLAN Notices', Vol. 41, ACM, pp. 16–29.

Zhang, Y., Prekas, G., Fumarola, G. M., Fontoura, M., Goiri, Í. and Bianchini, R. [2016], History-based harvesting of spare cycles and storage in large-scale datacenters, *in* 'Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation', number EPFL-CONF-224446, pp. 755–770.

Appendix A

French Résumé

A.1 Introduction

La dernière décennie a vu la prolifération de nombreux cadres distribués pour traiter une variété de l'analyse de données à grande échelle et des tâches de traitement. Tout d'abord, MapReduce [Dean and Ghemawat, 2008] été introduit pour faciliter le traitement des données en masse. Par la suite, des outils plus flexibles, tels que Dryad [Isard et al., 2007], Spark [Zaharia et al., 2012], Flink[1] et Naiad [Murray et al., 2013],pour nommer peu, ont été conçus pour répondre aux limites et la rigidité du modèle de programmation MapReduce. De même, des bibliothèques spécialisées telles que MLLib [Meng et al., 2016] et des systèmes comme TensorFlow [Abadi et al., 2016] ont vu la lumière pour faire face à des problèmes d'apprentissage machine à grande échelle. En plus d'un écosystème à croissance rapide, les cadres individuels sont guidés par un modèle de développement rapide, avec de nouveaux libère tous les quelques mois, introduisant des améliorations de performance substantielles. Depuis chaque cadre répond à des besoins spécifiques, un large choix d'outils et de combinaisons est disponible pour les utilisateurs qui peut aborder les différentes étapes de leurs projets d'analyse de données.

Ce contexte a suscité beaucoup de recherches [Delimitrou and Kozyrakis, 2013, 2014; Delimitrou et al., 2015; Ghit and Epema, 2016; Hindman et al., 2011; Isard et al., 2009; Kuzmanovska et al., 2016; Ousterhout et al., 2013; Schwarzkopf et al., 2013; Vavilapalli et al., 2013; Verma et al., 2015] dans la zone de l'allocation des ressources et des horaires, à la fois du milieu universitaire et de l'industrie. Ces efforts se matérialisent dans les systèmes de gestion de cluster qui offrent des mécanismes simples pour les utilisateurs de demander le déploiement de le cadre dont ils ont besoin. L'idée générale sous-jacente est celle du partage des ressources de cluster entre ensemble hétérogène de cadres, par opposition à la partition statique, qui a été rejeté pour cela entraîne une faible allocation des ressources [Hindman et al., 2011; Schwarzkopf et al., 2013; Verma et al., 2015]. Les systèmes existants divisent les ressources à différents niveaux. Certains d'entre eux, par exemple Mesos et YARN, ciblent l' orchestration de bas niveau des frameworks informatiques distribués: dans ce but, ils nécessitent des modifications de

[1] https://flink.apache.org/

ces cadres pour fonctionner correctement. D'autres, par exemple Kubernetes[2] et Docker Swarm[3], se concentrer sur l'approvisionnement et le déploiement des conteneurs, et sont donc inconscients des caractéristiques des cadres fonctionnant dans de tels conteneurs.

Malgré ces efforts, les ressources des centres de données sont souvent sous-utilisées, comme le montrent les traces récentes des déploiements de production à grande échelle [Reiss et al., 2012; Wilkes, 2011]: pour une charge de travail mixte les services de production et les applications par lots (voir la Figure 1.1); dans la plupart des cas (∼80%) l'utilisation des ressources est inférieur à 40% ou 80% des ressources allouées en fonction des différents types d'applications[4].

Les approches actuelles qui répondent aux exigences d'efficacité relèvent de deux grandes catégories. La première catégorie implique des méthodologies qui visent à diriger le comportement des locataires à travers la conception de l'incitation mécanismes; les locataires ont pour tâche d'optimiser le coût de fonctionnement de leurs applications, les fournisseurs opèrent sur les prix pour orienter l'allocation des ressources inutilisées. De telles approches sont largement adoptées par les fournisseurs de cloud public [Babaioff et al., 2017]. La deuxième catégorie concerne les approches qui fonctionnent au niveau du système, et proposer des mécanismes qui allouent des ressources en fonction des réserves des locataires[5][6] [Ghodsi et al., 2011; Hindman et al., 2011; Rasley et al., 2016; Schwarzkopf et al., 2013; Verma et al., 2015].

Le but ultime de la recherche ci-dessus est de rendre le concept de réservation de ressources obsolète, et soit laisser les locataires raisonner en termes de valeur et de coût [Babaioff et al., 2017], ou laissez le système déterminer comment éviter de gaspiller des ressources précieuses et coûteuses, surtout lorsque celles-ci sont rares et entraîner la mise en file d'attente des applications dans le planificateur.

Une contrainte de planification majeure que la recherche dans le domaine de l'allocation des ressources et de l'ordonnancement doit visage est sur la data-localité, qui se réfère à la capacité de déplacer le calcul près de l'endroit où le réel les données résident au lieu de déplacer de grandes données vers le calcul. Cela minimise la congestion du réseau et augmente le débit global du système. Alors qu'avant il était possible de mettre la tâche n'importe où, Maintenant, il doit aller sur l'une des répliques de données.

Cependant, de nos jours grâce à la virtualisation, les clusters de calcul et de stockage sont plus flexibles, ils peuvent être facilement approvisionné en différentes tailles et détruit lorsqu'il n'est pas nécessaire[7]. De plus en plus, un tel stockage et les systèmes de traitement sont exposés aux utilisateurs en tant que services , déployés sur le cloud computing public ou privé environnements, plutôt que sur des machines nues dans des clusters privés. En effet, de nombreuses entreprises proposent Clusters

[2] http://kubernetes.io/

[3] https://docs.docker.com/swarm/

[4] Dans l'analyse, nous avons vu que certaines applications utilisaient plus de ressources que demandé et cela a été confirmé par Google. Personnel. Leur système permet à l'utilisateur d'aller au-dessus de la réservation lorsque les ressources sont disponibles. Puisque tous les systèmes ne peuvent pas faire ceci (par exemple, Docker), nous avons décidé d'enlever cette partie des données

[5] http://www.docker.com/

[6] https://aws.amazon.com/emr/

[7] https://aws.amazon.com/application-hosting/benefits/

Analytics-as-a-Service (AaaS) pour exécuter une variété d'applications: Amazon Web Services (AWS) avec Elastic MapReduce[8], DataBricks Cloud[9], Cloudera Cloud[10] et Google Cloud Hadoop[11] exemples remarquables.

Dans les environnements de cloud computing, l'architecture des clusters d'analyse est le résultat de la de plusieurs services, composé de trois couches (séparées logiquement): la couche Calcul fait référence à les nœuds de cluster qui exécutent l'application de traitement de données (par exemple, une application Spark); la couche de données se réfère à toute combinaison de services de stockage (par exemple, HDFS[12] ou Swift[13]); et la couche de stockage physiquement stocke les données, y compris les disques éphémères, les magasins d'objets et de blocs élastiques.

En outre, il est probable que les couches de données ou de stockage et la couche de traitement se trouvent sur des racks différents ou même des centres de données: en conséquence, la sagesse traditionnelle de la localité de données peut être remise en question. Pour Par exemple, considérons Amazon S3[14]: les données résident sur un ensemble de machines dédiées uniquement au stockage, à la rupture localité de données complètement.

Actuellement, les utilisateurs de AaaS ont des informations abondantes sur les prix et sur la durabilité de Ressources. Il est possible de raisonner sur le dimensionnement des services basé sur les coûts et de sélectionner services de stockage en fonction de la disponibilité des données et des objectifs de durabilité. En conséquence, il est aujourd'hui possible de construire des pipelines d'ingestion de données, de stockage et de traitement, en composant - dans divers combinaisons - les trois couches définies ci-dessus.

Les questions que nous abordons dans cette thèse sont: qu'arrive-t-il à la performance, et au temps de réalisation en particulier, des applications d'analyse avec différents types de couche de calcul et de données configurations?

Au cours de notre quête pour répondre à cette question, nous découvrons l'existence d'un décalage d'impédance entre les frameworks à grande échelle et une solution de stockage de données, appelée stockage d'objets cloud, qui est actuellement largement utilisé parmi les fournisseurs; Amazon S3, Azure Blob storage[15], et IBM Cloud Object Storage[16], sont des systèmes de stockage en nuage distribué hautement évolutifs offrant une capacité élevée et un stockage rentable.

Jusqu'à présent, les connecteurs Hadoop pour le stockage des objets, par exemple, S3a[17] et Hadoop Swift Connector[18], été basé sur la sémantique des fichiers, une hypothèse naturelle étant donné que leur modèle de fonctionnement est basé sur le Hadoop interagit avec son système de stockage d'origine,

[8] https://aws.amazon.com/emr/
[9] Solution hébergée sur AWS: https://databricks.com/product/databricks-cloud
[10] Solution hébergée sur AWS: http://www.cloudera.com/content/cloudera/en/solutions/partner/Amazon-Web-Services.html
[11] https://cloud.google.com/hadoop/
[12] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[13] http://docs.openstack.org/developer/swift/development_saio.html
[14] https://aws.amazon.com/s3/
[15] https://azure.microsoft.com/en-us/pricing/details/storage/blobs/
[16] https://www.ibm.com/cloud-computing/products/storage/object-storage/cloud/
[17] https://aws.amazon.com/sdk-for-java/
[18] https://github.com/openstack/sahara-extra/tree/master/hadoop-swiftfs

Hadoop Distributed File System (HDFS). Cependant, traiter un stockage d'objets comme un système de fichiers constitue une inadéquation d'impédance, ce qui peut conduire à une mauvaise performance et une exécution incorrecte. En particulier, les opérations atomiques pour les fichiers ne peuvent pas être atomique pour les objets et les opérations qui sont peu coûteux pour les fichiers peuvent ne pas être peu coûteux pour les objets, et vice versa. Par exemple, renommer un répertoire dans un système de fichiers nécessite une seule opération atomique, tandis que dans le stockage d'objets, il nécessite des opérations de copie et de suppression pour chacun des objets de l'arbre sous le "répertoire virtuel"[19].

Nous ne sommes pas les premiers à reconnaître les mauvaises performances des connecteurs de stockage d'objets. D'autres ont essayé d'améliorer les performances, en sacrifiant l'exécution spéculative, puis en écrivant des objets directement à leurs noms définitifs, par exemple, DirectOutputCommitter[20] pour Amazon S3 ou en renommant la sortie Hadoop objets à leurs noms définitifs lorsque les tâches sont terminées (validation de la tâche) au lieu d'attendre le travail entier termine (travail de validation) [21]. Cependant, en raison de l'inadéquation de l'impédance, ces tentatives ont conduit à de subtils échecs

Les connecteurs actuels peuvent également entraîner des échecs et une exécution incorrecte, car l'opération de liste sur les conteneurs de stockage d'objets / seaux sont finalement cohérents. EMRFS[22] d'Amazon et S3mper[23] de Netflix surmontent la cohérence éventuelle en stockant des métadonnées de fichier dans DynamoDB[24], un fort supplément système de stockage cohérent séparé du magasin d'objets. Une fonctionnalité similaire appelée S3Guard[25] est en cours développé par la communauté open source Hadoop pour le connecteur S3a. Des solutions comme celles-ci, qui nécessitent plusieurs systèmes de stockage, sont complexes et peuvent introduire des problèmes de cohérence entre magasins. Ils ajoutent également des coûts puisque les utilisateurs doivent payer pour le stockage supplémentaire fortement cohérent.

Le but de cette dissertation est d'améliorer la réactivité des systèmes dans les clouds privés et publics fournisseurs de services, en appliquant de nouvelles techniques d'ordonnancement et en tirant parti des orienter le comportement des décisions d'ordonnancement. L'utilisation d'une telle approche améliore l'utilisation des ressources d'environ 50% et le délai d'exécution moyen, qui est le temps qu'une application réside dans le système, par plus de deux ordres de grandeur

A.2 Une heuristique de planification flexible

L'effort du **Chapter 3** est de combler le vide qui existe dans les approches actuelles et d' élever le niveau de abstraction à laquelle la programmation fonctionne. Nous introduisons une définition générale et flexible des applications , comment ils sont composés et comment les exécuter. Par exemple,

[19] Les magasins d'objets émulent les répertoires via une dénomination hiérarchique.
[20] https://github.com/apache/spark/pull/12229
[21] https://issues.apache.org/jira/browse/MAPREDUCE-6336
[22] https://aws.amazon.com/blogs/aws/emr-consistent-file-system/
[23] http://techblog.netflix.com/2014/01/s3mper-consistency-in-cloud.html
[24] https://aws.amazon.com/dynamodb/
[25] http://www.slideshare.net/hortonworks/s3guard-whats-in-your-consistency-model

une application utilisateur traitant de la formation d'un modèle statistique implique: un programme défini par l'utilisateur mettant en œuvre un algorithme d'apprentissage, un cadre (par exemple, Spark) pour exécuter un tel programme avec des informations sur ses besoins en ressources, le emplacement pour les données d'entrée et de sortie et éventuellement les paramètres exposés en tant qu'arguments d'application. Utilisateurs devrait être en mesure d'exprimer, de manière simple, comment une telle application doit être emballée et exécutée, soumettez-le et attendez-vous à des résultats dès que possible. Nous montrons que la planification de telles applications représente un écart par rapport à ce qui a été étudié dans la littérature sur les horaires, et nous présentons la conception d'un nouvel algorithme pour résoudre le problème. Un aperçu clé de notre approche est d'exploiter les propriétés de la les frameworks utilisés par une application, et distinguer leurs composants en fonction des classes, élastique: le premier étant requis pour qu'une application produise du travail, ce dernier contribuant à les temps d'exécution.

Notre heuristique concentre les ressources de cluster sur quelques applications et utilise la classe de composants d'application nents pour les emballer efficacement. Notre planificateur vise une allocation de cluster élevée et un système réactif. Il peut facilement accommoder une variété de politiques d'ordonnancement, au-delà du traditionnel «premier arrivé, premier servi» ou stratégies de «partage de processeurs», qui sont actuellement utilisées par la plupart des approches existantes. Nous étudions le performances de notre planificateur en utilisant des traces de charge de travail réalistes et à grande échelle de Google[26] [Reiss et al., 2012, 2011; Wilkes, 2011], et montrent qu'il surpasse constamment l'approche de référence existante qui ignore les classes de composants: les délais d'exécution des applications sont réduits de moitié et les files d'attente sont considérablement réduits. Cela induit moins d'applications en attente d'être servies, et augmente les ressources allocation jusqu'à 20% de plus que la base de référence. Enfin, nous construisons un système à part entière, appelé Zoé, qui programme des applications analytiques selon notre algorithme original et qui peut utiliser sophistiqué des stratégies pour déterminer les priorités d'application. Notre système expose une configuration simple et extensible langage qui permet la définition de l'application. Nous validons notre système avec des expériences réelles, et signaler des améliorations évidentes par rapport à un planificateur de base, lors de l'utilisation d'un représentant charge de travail: les délais d'exécution médians sont réduits jusqu'à 37% et l'allocation médiane des ressources est de 20% plus haute.

Le **Chapter 3** est organisé comme suit. Nous commençons par clarifier ce que sont les applications analytiques, donner exemples et formuler notre énoncé du problème dans la Section 3.1.Nous décrivons ensuite les détails de notre l'heuristique d'ordonnancement flexible, à la Section 3.2, que nous évaluons en utilisant des simulations à la Section 3.3. la mise en œuvre du système est décrite à la Section 3.4, et son évaluation est présentée à la Section 3.5.

[26] https://github.com/google/cluster-data

A.2.1 Définitions et énoncé du problème

Définitions

Nous définissons un framework d'analyse de données sous la forme d'un ensemble d'un ou de plusieurs composants logiciels (exécutable binaires) pour accomplir certaines tâches de traitement de données. Les frameworks distribués sont généralement composés par un contrôleur, un maître et un certain nombre de composants de travail. Exemples de frameworks distribués sont Apache Spark[27], Google TensorFlow[28] et MPI[29]. Un autre exemple d'analyse de données simple cadre que nous considérons est un Notebook interactif [Ragan-Kelley et al., 2014].

Les frameworks distribués nécessitent un ordonnanceur pour orchestrer leur travail: ils exécutent des jobs , chacun qui consiste en une ou plusieurs tâches exécutées en parallèle sur le même programme. Ces planificateurs fonctionnent à le niveau de la tâche : ils assignent des tâches aux travailleurs, et ils sont hautement spécialisés pour prendre en compte les particularités de chaque cadre.

Les ordonnanceurs de cadre tels que Mesos [Hindman et al., 2011] et Yarn [Vavilapalli et al., 2013] introduire un composant de planification supplémentaire pour partager des ressources de cluster entre des frameworks simultanés: Les règles de partage sont basées sur de simples variantes du partage de processeurs. De même, la gestion des clusters systèmes tels que Docker Swarm[30] et Kubernetes[31] utilisent un planificateur qui affecte des ressources génériques cadres. Le problème à résoudre est l' allocation efficace des ressources en plaçant le cadre les composants et leurs tâches sur les machines de cluster qui satisfont un ensemble de contraintes.

Nous sommes maintenant prêts à définir des applications analytiques, qui sont les éléments que nous programmons dans **Chapter 3**. Notre objectif principal est d' élever le niveau d'abstraction en manipulant une entité abstraite englobant un ou plusieurs cadres d'analyse, leurs composants et la logique nécessaire pour eux coopérer pour produire un travail utile en exécutant des tâches définies par l' utilisateur . Ce qui distingue notre travail de l'état de la technique est que notre planificateur prend en compte la notion de classes de composants, qui permet de modéliser la spécificité de chaque framework. Nous avons trouvé deux classes de composants distincts à suffisant pour modéliser les cadres analytiques existants: ainsi, les composants du cadre appartiennent soit à noyau ou à une classe élastique. Les composants de base sont obligatoires pour un cadre de travail utile; Au contraire, les composants élastiques contribuent à un travail, par exemple en réduisant son temps d'exécution. Considérer, par exemple, Spark. Pour produire du travail, il faut des composants de base: un contrôleur (le client spark) exécuter le planificateur DAG), un maître (dans un déploiement autonome) et un travailleur (exécutants exécutant). Nous traitons les travailleurs supplémentaires comme des composants élastiques. Un exemple alternatif est une application utilisant TensorFlow, qui ne fonctionne qu'avec les composants de base: un ou plusieurs serveurs de paramètres et un nombre des travailleurs. Ces deux frameworks

[27] http://spark.apache.org/
[28] https://www.tensorflow.org/
[29] https://www.open-mpi.org/
[30] https://docs.docker.com/swarm/
[31] http://kubernetes.io/

ont un comportement d'exécution sensiblement différent: Spark est un élastique cadre qui peut intégrer dynamiquement les travailleurs pour répartir les tâches. TensorFlow est rigide et utilise uniquement composants de base pour faire des progrès.

En résumé, la nature d'une application est celle d'élever le niveau d'abstraction et application est considérée comme étant une collection de cadres et de leurs composants hétérogènes en tant que entité unique à planifier et à allouer dans un groupe d'ordinateurs.

Déclaration de problème

Nous traitons maintenant les applications définies ci-dessus comme des entités abstraites que nous appelons des demandes : elles comprennent une ou plus de composants , appartenant à une classe donnée, soit noyau ou élastique. Dans la littérature, le classique le problème de la planification des demandes génériques devant être desservies par un système distribué a été étudié de manière approfondie [Dutot et al., 2004; Pruhs et al., 2004; Sgall, 2015]. Les demandes composées uniquement de composants de base sont généralement appelé rigide , tandis que les demandes composées uniquement de composants élastiques sont appelées malléable (si les ressources affectées sont décidées lorsque la demande est signifiée et qu'elles ne changent pas pour l'exécution entière) ou malléable (si les ressources peuvent varier pendant l'exécution [32]). Une différence clé en ce qui concerne les travaux antérieurs est que nous considérons des demandes hétérogènes , composées à la fois par le noyau et composants élastiques.

Pour simplifier l'exposition, nous supposons que les ressources système peuvent être mesurées en unités, et que il y a R unités disponibles globalement pour satisfaire les demandes. Chaque requête i spécifie le nombre d'unités pour ses composants de base et élastiques, étiquetésC_i et E_i respectivement. Idéalement, avec assez de disponible ressources, une requête est allouée à tous ses composants: dans ce cas, nous définissons le service (ou l'exécution) temps comme T_i. La quantité de travail pour satisfaire une requête est la surface du carré $W_i = T_i \times (C_i + E_i)$. Plus généralement, une requête est allouée à au moins $C_i + x_i(t)$ ressources, où $0 \leq x_i(t) \leq E_i$. Ensuite, le service le temps est $T_i' = \frac{W_i}{C_i + x_i(t)}$. Ce modèle simple permet de mettre à jour le temps de service T_i' lors d'une décision d'ordonnancement modifie $x_i(t)$, en mesurant la quantité de travail accomplie jusqu'à présent, et en calculant le quantité de travail à faire. Alors que des modèles plus complexes pour décrire T_i' peuvent être conçus, par exemple en tenant compte de la nature multidimensionnelle des ressources système ou des différents modèles d'évolutivité, notre simple approximation n'affecte pas la nature du problème d'ordonnancement que nous étudions. Dans ce qui suit, nous supposons que chaque demande peut pleinement utiliser le nombre spécifié de noyau et élastique composants, si des ressources leur sont accordées.

Essentiellement, le problème de la planification de l'exécution d'une charge de travail entrante est: i) trier les demandes pour décider dans quel ordre les servir; ii) allouer des ressources distribuées aux demandes sélectionné pour le service. La phase de tri peut être résolue en utilisant des approches naïves, par exemple la commande FIFO, ou des stratégies plus sophistiquées, qui utilisent des informations de

[32] Un exemple de cadre est malléable Spark [Zaharia et al., 2010]. Travailleur peut être ajouté ou retiré sans détruire l'exécution de l'application.

taille de requête. Encore plus généralement, les demandes peuvent être placés dans des "pools" et se voir assigner des priorités, pour imiter l'organisation hiérarchique des utilisateurs, pour Exemple. La phase d'allocation est plus délicate: dans l'abstrait, c'est un problème de "packing" qui détermine comment façonner les demandes traitées. Même en supposant que les temps de service soient connus a priori (par exemple, T_i est donnée en entrée), il est bien connu que le problème d'ordonnancement en ligne est NP-difficile [Pruhs et al., 2004]. Par conséquent, nous devons trouver une heuristique appropriée pour approximer une solution à l'optimisation de l'ordonnancement problème. Dans notre cas, cela revient à minimiser les délais d'exécution de l'application , qui est l'intervalle de temps entre la demande et la soumission. Dans le contexte que nous considérons, optimiser le délai d'exécution moyen représente une mesure de performance significative, car elle répond à la réactivité du système.

Notre problème d'ordonnancement ne prend pas directement en compte les contraintes de localisation des données. Comme nous l'avons vu dans [Pace et al., 2017], les fournisseurs de cloud récemment ont tendance à désagréger la couche de calcul et de stockage à différents niveaux: un nœud de calcul et de données peut résider sur le même hôte, sur des hôtes différents ou même sur des centres de données différents. Ensuite, nous motivons notre problème avec un exemple illustratif simple.

Exemple illustratif Nous considérons un système avec 10 unités de ressources disponibles, et quatre demandes en attente être servi, comme le montre la Figure A.1. Chaque demande nécessite 3 unités pour les composants de base, et différents unités pour les composants élastiques. Pour chaque requête, $T_i = 10$. Dans cet exemple, nous nous concentrons sur l'allocation phase seulement et nous utilisons la politique FIFO pour trier les demandes en attente.

Compte tenu de ces demandes, une approche traditionnelle et rigide de la planification - qui ne fait pas la distinction entre les classes de composants - assigne toutes les ressources requises à chaque requête. Puisque toutes les demandes ont besoin au moins 5 unités ($C_i + E_i \geq 5$), et puisque toute paire de demandes a un besoin cumulé supérieur à 10 unités, le planificateur sert une requête à la fois (Figure A.1, en haut): le délai d'exécution moyen est de 25s. Remarque que, dans ce cas, le remblayage n'est pas possible, c'est-à-dire même en changeant l'ordre dans lequel les demandes sont traitées la situation ne change pas.

Une autre approche d'ordonnancement vient de la littérature de la planification des travaux malléable. Le planificateur affecte toutes les ressources à la première demande dans la ligne d'attente, puis affecte les ressources restantes (si any) à la demande suivante, et ainsi de suite, jusqu'à ce qu'il n'y ait plus de ressources disponibles. Cette heuristique a été montré être proche de l'optimum [Dutot et al., 2004]. LA Figure A.1,au milieu, illustre l'idée: demande B peut être servi avec la requête A. Lorsque la requête A est terminée, le planificateur affecte d'abord ressources pour demander B, puis essaie de répondre à la demande suivante. De même, lorsque la demande B est terminée, le planificateur affecte d'abord plus de ressources pour demander C, puis tente de servir la requête D. Cependant, puisque la requête D nécessite au moins $C_i = 3$ unités, le planificateur est bloqué (notez que la requête C utilise 8 unités), donc la requête D doit attendre et certaines ressources système restent inutilisées. Le délai d'exécution moyen est de 20 secondes.

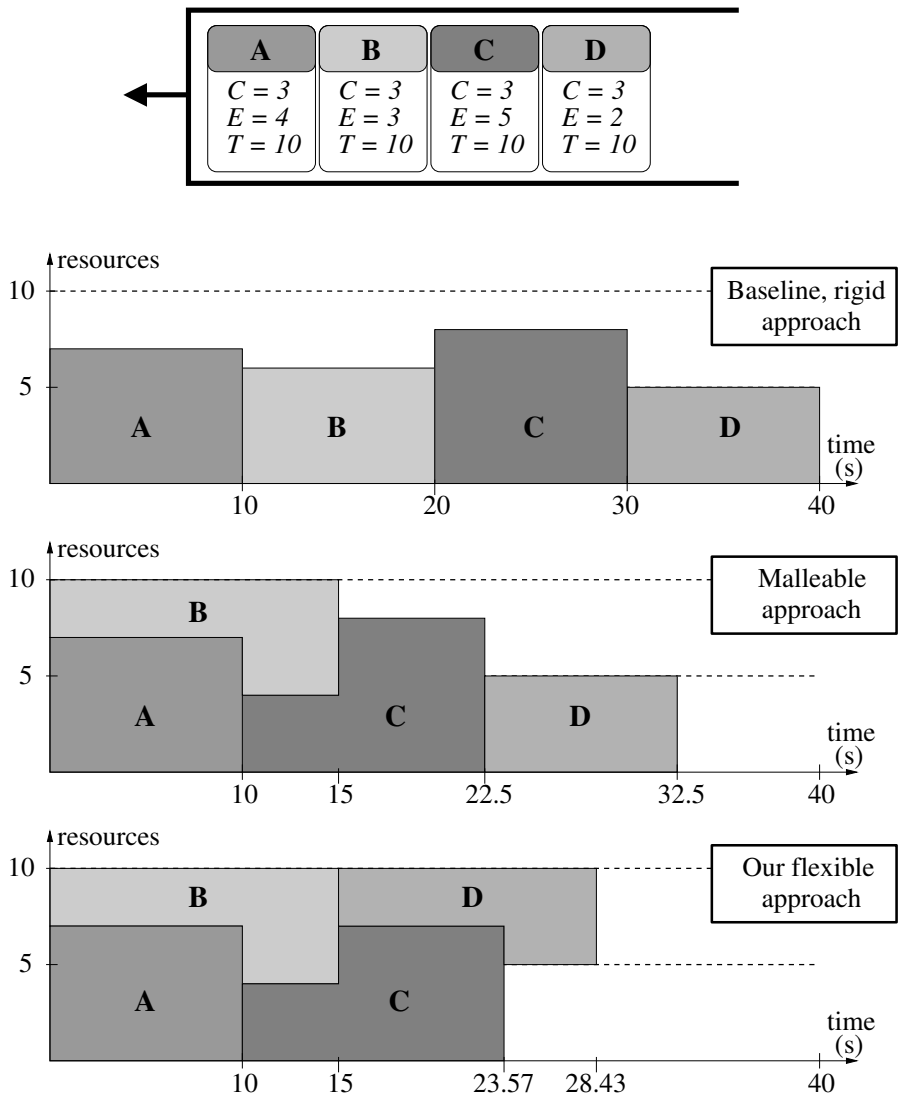


Fig. A.1 Exemples d'ordonnancement des demandes: (haut) rigide, (moyen) malléable, (bas) flexible approches.

Nous préconisons la nécessité d'une nouvelle approche de la planification, qui distingue les classes de composants. L'idée est d'exploiter la flexibilité des composants élastiques et d'utiliser plus efficacement les ressources du système. Intuitivement, une solution aux problèmes des heuristiques existantes consiste à récupérer certaines ressources assignées à composants élastiques d'une requête en cours et les affecter à une requête en attente. Ceci est montré dans le en bas de la Figure A.1: le planificateur récupère juste une unité de la requête C pour qu'elle puisse fournir 3 unités pour demander D, qui sont suffisantes pour démarrer ses composants de base et produire un travail utile. Avec ça approche, le redressement moyen est de 19.25s.

Bien que la solution ci-dessus semble simple, elle pose de nombreux défis: combien d'unités assignées à les composants élastiques peuvent être sacrifiés pour servir la prochaine requête? Combien

de demandes devraient être servies simultanément? Si le planificateur se concentre uniquement sur les composants de base, assurez-vous que de nombreuses demandes sont servi simultanément? Comment la planification peut-elle prendre en compte les priorités assignées par la phase de tri?

Le dernier point introduit un défi supplémentaire, lié aux politiques d' ordonnancement préemptives . Si un haut demande de priorité arrive, car il n'est pas possible d'interrompre les composants de base - car cela tuerait le demande - comment pouvons-nous sélectionner et préempter les composants élastiques pour répondre à la nouvelle demande?

Étant donné des demandes composites hétérogènes, qui ne sont ni rigides, ni malléables (mais les deux), disponibles les heuristiques d'ordonnancement dans la littérature ne répondent pas aux problèmes de tri et d'allocation: une nouvelle approche est donc vraiment souhaitable.

A.2.2 Un algorithme de planification flexible

Nous caractérisons une demande par son heure d'arrivée, sa priorité (pour décider de l'ordre dans lequel les demandes doivent être servi), les ressources demandées (noyau et élastique) et le temps d'exécution (en isolement, c'est-à-dire quand tout ressources requises sont accordées à l'application). Compte tenu de la charge de travail entrante, notre objectif est d'optimiser la somme des temps d'exécution τ_i, c'est-à-dire:

$$\min \sum_i \tau_i \Rightarrow \min \sum_i (\text{queuing}_i + \text{execution}_i)$$

Le temps d'exécution réel dépend de la quantité de ressources affectées au fil du temps à la demande. À présent, rappelons que le problème d'ordonnancement peut être divisé en phases de tri et d'allocation. Le tri détermine lorsqu'une requête est traitée, cela a un impact sur son temps de mise en file d'attente. La phase d'allocation contribue à la fois à la file d'attente et les temps d'exécution réels. Selon la granularité de l'allocation [Schwarzkopf et al., 2013], une requête peut avoir besoin d'attendre qu'un certain nombre de ressources soient disponibles avant de les occuper, augmenter - quoique indirectement - le temps d'attente. Le temps d'exécution est directement lié à l'allocation algorithme et aux caractéristiques de la charge de travail.

Nous découplons le tri des requêtes de l'allocation:[33]notre ordonnanceur maintient l'ordre de requête, comme imposé par un composant externe, et se concentre uniquement sur l'allocation des ressources. Le tri peut être simplement en fonction des heures d'arrivée (ce qui revient à implémenter une discipline de file d'attente FIFO), ou peut utiliser des l'information, comme la taille de la demande (mettant ainsi en œuvre une variété de disciplines fondées sur la taille).

Dans l'ensemble, nous optimisons les délais d'exécution des demandes grâce à une allocation judicieuse des ressources et à la conception un algorithme qui s'efforce d'allouer toutes les ressources de cluster disponibles, en servant le moins de demandes à la fois . Intuitivement, en concentrant les ressources sur quelques requêtes, nous attendons leur temps d'exécution être petit. Par conséquent, les

[33]Cette approche est similaire à celle utilisée dans l'ordonnanceur SLURM [Yoo et al., 2003], où l'ordre des travaux en attente est donné par un composant externe, connectable, et le planificateur traite les travaux suivant cet ordre.

requêtes mises en file d'attente bénéficient également de temps d'attente réduits, car les ressources sont libérées plus vite.

A.2.3 Résumé

La gestion efficace des ressources des grappes d'ordinateurs a été un domaine de recherche de longue durée, avec des pics de l'attention se produit en conjonction avec des améliorations dans les machines de calcul, par exemple récemment avec le nuage informatique et big data. Une nouvelle génération de systèmes de gestion de clusters, visant à devenir "centre de données" systèmes d'exploitation, sont actuellement confrontés à des problèmes d'efficacité et de performance à grande échelle.

Malgré les progrès récents, il existe un écart entre l'objectif de gestion des ressources de bas niveau, et celle de la manipulation d'applications de haut niveau, hétérogènes, distribuées (analytiques) fonctionnant environnements de cluster. Au **Chapter 3** nous avons présenté une première étape possible pour combler cette lacune, sous la forme d'un nouveau planificateur d'applications qui interagit avec un backend de gestion de cluster, pour planifier et allouer des ressources à des applications définies avec un langage et une sémantique simples. En plus de soigner l'ingénierie, nécessaire pour concevoir et mettre en œuvre notre système que nous appelons Zoe, notre recherche a identifié un plus problème fondamental, qui nécessite une nouvelle heuristique d'ordonnancement capable de manipuler composite applications, tout en contribuant à la réactivité du système.

Nous avons validé notre algorithme pour résoudre notre problème d'ordonnancement selon deux axes. Nous avons utilisé un numérique approche pour simuler des déploiements et des charges de travail à grande échelle. Merci à plusieurs numériques différents expériences (voir la Section 3.3) nous avons montré que notre algorithme de planification était très efficace pour réduire les délais d'exécution, en particulier en réduisant les temps de mise en file d'attente des applications. Par conséquent, les ressources du cluster étaient mieux répartis.

En outre, nous avons signalé (voir la Section 3.5) un aperçu de l'évaluation de Zoe, qui indique performances et efficacité supérieures liées à notre heuristique d'ordonnancement flexible.

A.3 Allocation de ressources basée sur les données

Au **Chapter 4**, nous discutons d'une méthodologie qui se situe essentiellement dans la deuxième catégorie. Nous présentons un système qui alloue dynamiquement des ressources en fonction des observations historiques de son utilisation. Plus plus précisément, nous utilisons des algorithmes d'apprentissage automatique pour ajuster l'allocation aux lisation. Nous présentons notre conception d'un mécanisme d'ordonnancement piloté par les données qui améliore l'utilisation des clusters, réduisant ainsi le délai d'exécution moyen, tout en empêchant les défaillances d'application dues aux ressources contention. Notre approche surveille l'utilisation des ressources et s'appuie sur des ressources en ligne sophistiquées la prévision de la demande pour moduler les ressources allouées, telles qu'elles permettent une approximation des schémas d'utilisation. Nos expériences, que nous menons sur un

simulateur de système ainsi que d'un prototype de mise en œuvre en utilisant traces de centre de données réelles, indiquent des gains substantiels par rapport aux alternatives existantes: notre approche contribue à des clusters plus efficaces et réactifs, tout en contrôlant soigneusement le nombre de défaillances d'applications en raison de la nature approximative de notre approche de contrôle.

Le reste du **Chapter 4** Le reste du Dans la section 4.2 nous présentons la conception de notre système, et nous validons nos idées en utilisant une campagne de simulation dans la section 4.3. Enfin, nous présentons notre prototype mise en œuvre à la section 4.4 et son évaluation à la section 4.5.

A.3.1 Énoncé du problème

Dans cette thèse, nous étudions le problème de l'efficacité de la grappe en réduisant le relâchement des ressources induit par ordonnanceurs d'application centrés sur la réservation, qui correspondent à l'allocation à la réservation. Pour ce faire, nous présentons un nouveau mécanisme qui prédit l'utilisation des ressources et ajuste l'allocation des ressources en conséquence. Le principal défi à relever est que les erreurs de prédiction peuvent avoir des conséquences problématiques, puisque les pointes pourraient faire des ravages dans le système [Verma et al., 2015].Lorsque vous traitez avec des ressources finies telles En effet, la RAM, en ne fournissant pas la bonne quantité de ressources, conduit à des échecs d'application. Prudent l'ingénierie suggérerait d'introduire un tampon qui servira de «garde-fou» aux erreurs de prédiction. Ce il en résulte un compromis, puisque d'une part le tampon de sécurité doit être petit pour minimiser le jeu, D'autre part, il devrait être suffisamment grand pour éviter les échecs d'application.

Les travaux antérieurs (une description détaillée est fournie à la Section 2.1.3) sont généralement considérés comme partageables ressources, telles que CPU, où l'effet du mauvais dimensionnement des ressources ne se traduit pas en échecs d'application. D'autres approches considèrent le surapprovisionnement en ressources, où le relâchement n'est pas continuellement optimisé, et où les échecs d'application peuvent être imprévisibles et sont pris en charge par le Operating System (OS).

Dans notre approche, nous tirons parti de trois idées clés: la confiance de prédiction, l'élasticité de l'application et échecs contrôlés. Dans le processus de prédiction, la plupart des outils fournissent des informations supplémentaires sur la confiance de la prédiction. Nous utilisons ces informations pour adapter dynamiquement le tampon de sécurité cela devrait empêcher les échecs de l'application. En outre, les cadres, sur lesquels les applications sont basés, sont composés de plusieurs éléments qui sont caractérisés par un noyau ou un caractère élastique [Pace, Venzano, Carra and Michiardi, 2016]. Les composants de base sont obligatoires pour un cadre de production par exemple, Apache Spark nécessite un contrôleur, un maître et un ouvrier); composants élastiques, à la place, contribuer de manière optionnelle à un travail, par exemple en réduisant son temps d'exécution. Une application qui comporte seuls les composants de base sont appelés rigides, tandis que les applications avec un mélange de composants de base et d'élastiques sont appelés élastiques. Si la demande en ressources est supérieure aux ressources disponibles, nous intervenons possible) sur les composants élastiques pour éviter les défaillances d'application. En dernier lieu, les deux précédents devraient-ils mécanismes ne suffisent

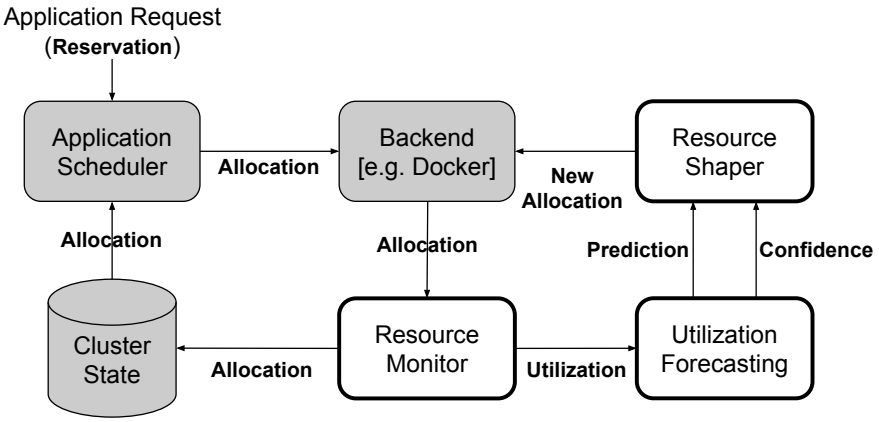


Fig. A.2 Vue d'ensemble du système: les boîtes ombrées représentent les composants existants, les boîtes blanches indiquent les nouvelles composants présentés dans ce travail.

pas à fournir suffisamment de ressources, nous décidons explicitement quelle application échouer afin de minimiser la quantité de travail gaspillé.

A.3.2 Conception du système

L'objectif de notre système est d'augmenter l'utilisation des clusters et de réduire les délais d'exécution moyens en ajustant l'allocation pour suivre l'utilisation des ressources en anticipant sa dynamique, tout en réduisant la nombre d'échecs d'application «auto-infligés» dus à des erreurs d'approximation.

La Figure A.2 illustre l'architecture que nous supposons dans notre travail. Le module backend est une instance de un système de gestion de cluster, tel que Docker[34] ou Kubernetes[35], ou des ordonnanceurs alternatifs [Thinakaran et al., 2017]. De plus, nous supposons la présence d'un planificateur d'application tel que [Pace, Venzano, Carra and Michiardi, 2016], qui lit l'état de cluster de calcul à partir d'un composant de base de données dédié. Enfin, le composant de surveillance remplit la base de données d'état du cluster avec des mesures prises à partir de le backend. Dans cette section, nous nous concentrons sur les deux composants supplémentaires que nous présentons dans cet article: module de prévision d'utilisation , et le module de mise en forme de ressources .

La vue d'un oiseau sur le fonctionnement de notre système est la suivante. Les demandes d'exécution d'application prennent la forme des réservations de ressources, qui sont soumises au planificateur d'application. L'application le planificateur admet la requête sur la seule base des informations de réservation et ordonne au back-end de fournir les ressources nécessaires. Le moniteur de ressources recueille des informations sur les deux alloués et les ressources utilisées, qui sont respectivement transmises à l'état du système et à la composante de prévision. Le module de mise en forme des ressources évalue l'allocation des ressources pour correspondre aux modèles d'utilisation prévus, et

[34] https://docs.docker.com/swarm/
[35] http://kubernetes.io/

est responsable de la préemption des applications en cours en cas de pics soudains dans la demande de ressources. L'allocation de ressource modifiée est reflétée dans l'état du système, ce qui déclenche à son tour une nouvelle planification les décisions. Ensuite, nous décrivons en détail les composants qui matérialisent nos idées.

Resource monitor Ce module recueille des informations sur l'allocation et l'utilisation des ressources chaque composant de chaque application en cours d'exécution. Cela se produit à des intervalles de temps réguliers: fréquences plus élevées fournir des vues plus précises, mais générer plus de données. Notre objectif est de minimiser l'intrusion par étant agnostique d'application: pour cette raison, nous n'instruisons pas d'applications (comme cela a été fait par exemple dans [Kuzmanovska et al., 2016]), mais prenez des métriques standard (CPU, mémoire, etc.) telles qu'elles sont vues par le Operating System (OS).

Utilization forecasting Le but de ce module est d'anticiper l'utilisation des ressources de chaque composant d'application. Nous étudions les approches de modélisation paramétrique et non-paramétrique pour prédire l'utilisation des ressources, en mettant l'accent sur la quantification de l'incertitude associée à ces prédictions. Un exposé plus détaillé de la méthodologie que nous employons se trouve à la Section 4.2.1.

Resource shaper Ce module utilise des prévisions d'utilisation pour ajuster les ressources allouées à chaque composant des applications en cours d'exécution. Nous anticipons les erreurs de prédiction, donc nous compensons en utilisant un garde tampon de taille β pour augmenter artificiellement (c'est-à-dire pour forcer l'estimation) la ressource de pointe prédite utilisation. Un exposé plus détaillé de β peut être trouvé à la Section 4.2.1.

De plus, le formateur de ressources est en charge de la préemption des applications. Les politiques de préemption peuvent Soit être optimiste [Schwarzkopf et al., 2013; Verma et al., 2015] ou strict (pessimiste). Nous défendons une politique stricte, pour éviter de déléguer la préemption des applications à l'OS, qui gère la pénurie de ressources (comme Out Of Memory (OOM)) dans une application agnostique et "imprévisible" façon. Un détaillé L'exposé de la politique de préemption peut être consulté à la Section 4.2.2.

A.3.3 Résumé

L'émergence du paradigme «le centre de données en tant qu'ordinateur» a conduit à des avancées sans précédent les cadres de gestion de cluster, qui visent à exposer des ressources de cluster distribuées à une variété de applications critiques et scientifiques. Cependant, le modèle actuel de réservation de ressources empêche un utilisation efficace des ressources du cluster. La dynamique d'utilisation des ressources induit un surprovisionnement, ce qui du principal coupable de mauvaise efficacité. Le problème de la sous-utilisation a été résolu par plusieurs approches. Par exemple, la conception

d'incitations économiques pour orienter le fonctionnement du système a conduit à le développement de marchés de ressources complexes, par exemple les instances AWS Spot, qui nécessitent l'échec de la conception applications tolérantes, en raison de la nature éphémère des ressources qui leur sont offertes.

Dans le **Chapter 4**, nous avons présenté un mécanisme qui coopère avec un planificateur pour ajuster dynamiquement ressources allouées à une application, de sorte qu'elles correspondent étroitement à celles qu'elles utilisent cycle de la vie. Notre conception comportait: une méthode pour construire un modèle statistique pour prévoir l'utilisation des ressources, et une politique de préemption qui réalloue les ressources système tout en minimisant les pannes.

Nous avons validé numériquement notre mécanisme et avec une vraie campagne expérimentale. Notre simula- tions éclairent le rôle clé joué par notre capacité à modéliser et utiliser l'incertitude de la prévision, et par le utilisation de la préemption stricte par rapport au contrôle optimiste de la concurrence. Nous avons mis en place un prototype de système de notre mécanisme d'allocation dynamique et déployé dans un environnement de test, où nous avons exécuté une véritable charge de travail. Les résultats indiquent notamment une amélioration de l'efficacité du système, qui se traduit par une meilleure réactivité.

A.4 Évaluation expérimentale de la désagrégation entre calcul et Espace de rangement

Le question que nous abordons au **Chapter 5** est: qu'arrive-t-il à la performance, et à l'achèvement temps en particulier, des applications d'analyse avec différents types de configurations de couche de calcul et de données?

Nous adoptons une approche expérimentale, et proposons une méthodologie et une campagne de mesure, dont L'objectif est d'analyser la performance correspondant à une notion intuitive de distance entre le calcul arrive et les données résident. Ce faisant, nous définissons un ensemble complet de charges de travail applicatives cela remet en question les systèmes étudiés de différentes manières. En fin de compte, notre objectif est de surmonter limitations des travaux antérieurs qui fournissent seulement une vision booléenne de la localité de données: nos résultats indiquent que - en général - la mesure de distance intuitive que nous présentons dans ce travail est un bon indicateur de la raison à propos du classement des performances. Cependant, l'inadéquation d'impédance entre différents services et application les charges de travail doivent être prises en compte pour formuler des explications plausibles pour les valeurs aberrantes en termes de performance.

En particulier, nous montrons que la localité de données ne peut pas être étudiée comme une caractéristique qui peut être présente ou non, comme cela arrive de nos jours, mais qu'il existe différents degrés de localité de données, dont l'importance varie en fonction de la configuration de l'infrastructure d'analyse de données et de la charge de travail spécifique. En outre, nous soulignons certains problèmes de conception d'une architecture de service de stockage spécifique (Swift) cette solution n'est pas

optimale pour l'exécution d'une structure d'analyse de données; et nous montrons l'impact de la mise en cache données chaudes au niveau de la couche de données par rapport à l'exécution de l'application.

Le **Chapter 5** est divisé en différentes parties: la Section 5.1 explique notre question de recherche, la Section 5.2 décrit la méthodologie et la Section 5.3 est consacrée aux résultats.

A.4.1 Énoncé du problème

Nous évaluons empiriquement la performance des applications analytiques composées à l'aide de divers calculs, Configurations de couche de données et de stockage. Notre objectif est de comprendre comment le temps d'exécution de l'application varie configurations, pour un large éventail de charges de travail d'application.

La modélisation des performances des systèmes distribués complexes est une tâche ardue: l'exécution de l'application est affectée par plusieurs facteurs, y compris la localité de données (qui est la base du traitement parallèle des cadres tels que Hadoop et Spark), l'inadéquation de l'impédance entre les différents services impliqués applications analytiques, interférences entre locataires concurrents, charges de travail applicatives et bien d'autres.

A ce titre, nous adoptons une approche expérimentale et analysons la performance des applications à travers les lentilles du principe de localité de données, que nous revisitons pour accommoder les configurations de souffle de stockage actuellement disponible dans la plupart des nuages publics et privés. Dans notre étude, nous soulignons les problèmes qui se posent une conséquence de la composition du service et suggère des moyens de les atténuer.

A.4.2 Scénarios de déploiement

Nous définissons un scénario de déploiement comme une configuration de couches de calcul, de données et de stockage et étudions 4 scénarios que nous pensons représentatifs des configurations communes:

Guest Collocation (GC) Les couches de calcul et de données sont hébergées sur la même machine virtuelle (VM), la couche de stockage est un disque local éphémère. Ceci est une configuration populaire dans les nuages publics: quand le cluster GC est mis hors service, toutes les données sont perdues.

Guest Collocation with Volumes (GC-V) Même configuration que pour GC, mais la couche de stockage utilise volumes provisionnés à l'aide du système de fichiers distribués Ceph. ceci est également une configuration populaire dans nuague publics: il permet l'élasticité à la couche de calcul, sans sacrifier la durabilité des données au niveau des données et couches de stockage.

Swift (SWI) Les couches de calcul et de données s'exécutent sur des hôtes distincts. La couche de données est l'objet Swift magasin, ce qui assure la durabilité des données. De même que pour la configuration GC-V, ce scénario permet de calculer élasticité de couche, et c'est une configuration populaire pour son interface REST simple d'interagir avec des données.

No Collocation (NC) Les couches de calcul et de données s'exécutent sur des hôtes différents; la couche de données utilise HDFS monté sur une couche de stockage qui utilise un disque local éphémère. Ceci est un scénario permettant la durabilité des données: la couche Compute peut être mise hors service, tandis que la couche Data and Storage continue de s'exécuter.

Comme nous l'expliquons à la section 5.2.3, chaque scénario présente un degré de localisation des données différent. En outre, nous notons un premier exemple de décalage d'impédance, que nous développons plus loin dans le reste de cette travail. En effet, les couches Data et Storage peuvent implémenter leur propre mécanisme de réplication de données. Cela est évident dans le scénario GC-V: lorsque HDFS est monté sur des volumes, la réplication HDFS et Ceph les mécanismes sont redondants. Pour mieux comprendre les implications de performance du scénario GC-V, on distingue ainsi deux cas: GC-Vb, avec réplication HDFS et Ceph et GC-V, avec seulement Réplication Ceph, qui correspond aux degrés de liberté exposés aux utilisateurs pour la configuration de leur prestations de service. Notez que si la réplication HDFS est désactivée, Spark (la couche de calcul) n'a aucun autre moyen de récupérer les données si un datanode tombe en panne, entraînant une défaillance de l'application; aussi le planificateur de Spark a moins de flexibilité dans la planification des tâches, car les blocs de données sont présents dans un seul noeud de données. En revanche, En activant la réplication HDFS, l'application écrira des données supplémentaires.

A.4.3 chemin Compute-to-Data

Nous introduisons une notion intuitive de la distance entre le lieu de calcul et la donnée. Comme exemples illustratifs, considérons les cas suivants: calcul et les données résident sur la même machine virtuelle, sur différentes machines virtuelles s'exécutant dans le même hôte physique, sur différentes machines virtuelles sur différents hôtes physiques dans le même rack, et ainsi de suite.

Il est intuitif de traiter ces cas comme augmentant en termes de distance, ce qui est donc faiblement couplé avec la quantité de liens réseau que les données doivent parcourir pour être traitées. Aussi, rappelez-vous que lire et écrire des opérations émises par la couche de calcul, travailler sur une division donnée des données d'entrée ou de sortie, qui est organisé comme une séquence d'enregistrements. Nous utilisons la définition intuitive et approximative de distance suivante:

Definition *Le chemin Compute-to-Data est le nombre de liens logiques réussis (en lecture ou en écriture) l'opération doit croiser, pour un enregistrement de données donné.*

Nous utilisons le chemin Compute-to-Data comme un proxy pour raisonner sur le classement des performances, qui prend encompte la distance logique entre les trois couches composant un service analytique et le coût du (des) système (s) de réplication. En effet, on peut s'attendre à une dégradation des performances chaque fois qu'un opération traverse un lien réseau: le classement intuitif est valable même si nous ne modélisons pas explicitement le réseau latence ou topologie. Notez qu'il est important d'être prudent et de prendre en compte les détails de l'architecture de chaque couche: par exemple, Swift a un point d'accès unique appelé Swift-Proxy, qui sert de médiateur entre la couche de calcul et les nœuds de stockage de Swift.

Pour calculer le chemin Compute-to-Data, nous comptons tous les liens logiques, entre la couche Computeet les données physiques, que chaque demande d'enregistrement individuelle doit traverser. Considérant le scénario SWI, nous avons un lien de la couche de calcul à Swift-proxy, puis un de plus entre Swift-proxy et Swift ' nœuds de stockage, les ACK de la requête d'enregistrement traversent les mêmes liens pour atteindre la couche Compute, par exemple un total de 4 liens. Une considération similaire peut être faite pour les scénarios restants: le GC traverse 0 lien tandis que les liens GC-V et NC 2.

Pour les demandes d'écriture, la réplication de données doit être prise en compte. HDFS, Swift et Ceph ont Différents systèmes de réplication: HDFS utilise la réplication en chaîne , tandis que Swift et Ceph utilisent l' asynchrone réplication , avec des quorums différents. En supposant un facteur de réplication de 3, le proxy Swift nécessite 2/3 des nœuds de stockage pour accuser réception d'une opération d'écriture, alors que Ceph nécessite tout le stockage d'objets Démons (OSD) pour reconnaître le succès. Les Figures 5.1 and 5.2 illustrent comment calculer le chemin Compute-to-Data pour chaque scénario lors d'une requête de lecture et d'écriture.

Le chemin Compute-to-Data pour les opérations de lecture et d'écriture et pour différents scénarios est résumédans le table 5.1. Dans le tableau, nous organisons et classons les scénarios en fonction de leur distance: intuitivement, nous attendonsperformances de l'application pour suivre le même classement que nous produisons en utilisant le chemin Compute-to-Data . Notre Les résultats de mesure indiquent que le chemin Compute-to-Data est un bon proxy pour classer les scénarios en fonction deleur performance relative attendue, bien que l'intuition ne soit pas suffisante seule pour expliquer ce que nous soutenons avec des données.

A.4.4 Résumé

Choisir la bonne composition des services analytiques est un problème difficile, impliquant des considérations de coût, les exigences de durabilité des données et, finalement, les performances attendues des applications. Notre expérimental les résultats ouvrent la voie à des décisions éclairées sur les déploiements AaaS. Dans la suite, nous résumons nos résultats et leurs implications.

Composition du service Une configuration qui vise à atteindre la durabilité des données malgré l'éphémère La nature des machines virtuelles et les services qu'elles exécutent doivent être conçus avec soin. Pour des raisons allant de la facilité de l'intégration à la familiarité avec des API bien établies, il est tentant de composer des services comme dans le NC (pas de collocation) scénario que nous étudions dans ce travail. Nos résultats montrent que c'est un mauvais choix pour un large gamme de charges de travail, dans laquelle les cycles de CPU précieux sont perdus pour attendre que les données circulent sur le réseau

Volumes L'utilisation de volumes provisionnés au-dessus d'un système de fichiers distribué tel que Ceph est étonnamment performante bien. Ceci est inattendu car, de même que le scénario CN, le réseau

est fortement impliqué l'exécution de l'application. Cependant, nos résultats indiquent que même avec une bande passante de bissection modeste, le Couche de calcul peut faire des progrès rapides vers la fin d'une application, grâce à l'efficacité de rayure.

Cependant, à titre de mise en garde, nos résultats indiquent une inadéquation potentielle de l'impédance entre Les couches de stockage, en raison de l'interaction de plusieurs mécanismes de réplication. En tant que tel, les utilisateurs finaux devraient soyez conscient de la situation et configurez la couche de données de façon appropriée, de sorte que la réplication des données ne effectuée par la couche de stockage, car cela est très bénéfique pour les performances de l'application.

De plus, nos résultats indiquent que les fournisseurs de cloud computing pourraient différencier leur volume offre: les volumes à usage général fonctionneraient comme d'habitude, alors que les volumes analytiques devraient désactiver les données réplication Dans ce cas, les utilisateurs finaux contrôleraient totalement la réplication: nos résultats montrent que - en particulier pour les charges de travail intensives en écriture - cela produit des performances supérieures.

Swift La performance de Swift est décevante. Ceci est dû à une autre instance d'impédance incompatibilité entre la couche Calcul et Données. Impossible de renommer rapidement les fichiers sans créer réellement une nouvelle copie, entraîne de graves pénalités de performance, faisant de Swift une solution sous-optimale.

En outre, nous notons que l'architecture Swift a été conçue pour des applications très différentes à partir de cadres de calcul parallèles: nos résultats indiquent que le proxy Swift peut représenter un goulot d'étranglement, car il est impliqué dans le chemin de données. Certes, l'utilisation d'un serveur proxy comme coordinateur permet le cluster gestionnaires d'ajouter facilement des flux de contrôle à Swift, mais cela dégrade les performances. Une solution qui est actuellement adopté par plusieurs sociétés utilisant Swift, au niveau de la production, est d'ajouter plusieurs procurations et équilibrer la charge de trafic entre eux: mais parce que la charge de travail peut changer de façon imprévisible, un puits plan de capacité de pensée n'est pas facile à obtenir. Comme souligné précédemment, plus de proxies rendront les Swift architecture en quelque sorte similaire à HDFS, mais à des coûts plus élevés. Travaux récents d'IBM [Rabinovici-Cohen et al., 2014] montre que certains flux de contrôle et les transformations de données peuvent être faites beaucoup plus près de la nœuds de stockage. En conséquence, il est tentant de suggérer la conception d'un nouveau proxy Swift qui pourrait se comporter de la même façon que le nom de domaine HDFS: un tel proxy alternatif n'agirait que comme un stockage de métadonnées, et ne serait pas impliqué dans le chemin de données réel.

Caching Enfin, nos résultats montrent que la mise en cache joue un rôle important dans la détermination de l'application performance. D'une part, la mise en cache "interrompt" le chemin Compute-to-Data qui peut être déduitàpartir des opérations de lecture / écriture sur les enregistrements de données, ce qui rend la performance de l'application plus difficile à prédire. D'un autre côté, en réduisant le chemin Compute-to-Data , il atténue les problèmes de plusieursconfigurations que nous avons étudiées, ce qui

est utile pour les utilisateurs finaux car il donne plus de flexibilité dans le choix Couches de données et de stockage.

Cependant, la conception du mécanisme de mise en cache inter-application pour les frameworks de traitement en parallèle est encore à ses balbutiements: Tachyon [Li et al., 2013] et HDFS2 sont de bons exemples d'approches récentess'attaquer à ce problème.

A.5 Stocator: connecteur haute performance pour les magasins d'objets for Object Stores

Dans le **Chapter 6** nous présentons Stocator[36], un connecteur de stockage haute performance, qui permet à Hadoopmoteurs d'analyse pour travailler directement sur des données stockées dans des systèmes de stockage d'objets. Ici, nous nous concentrons sur Spark Cependant, notre travail peut être étendu pour travailler avec les autres parties de l'écosystème Hadoop.

Jusqu'à présent, les connecteurs Hadoop pour le stockage des objets, par exemple S3a[37] et Hadoop Swift Connector[38], été basé sur la sémantique des fichiers, une hypothèse naturelle étant donné que leur modèle de fonctionnement est basé sur le façon dont Hadoop interagit avec son système de stockage d'origine, HDFS. Cependant, traiter le stockage d'objets comme un système de fichiers constitue une incompatibilité d'impédance, ce qui peut conduire à une mauvaise performance et incorrecte exécution. En particulier, les opérations atomiques pour les fichiers peuvent ne pas être atomiques pour les objets et les opérations qui sont peu coûteux pour les fichiers peuvent ne pas être peu coûteux pour les objets, et vice versa. Par exemple, renommer un répertoire dans un système de fichiers nécessite une seule opération atomique, alors que dans le stockage d'objets, il nécessite une copie et supprimer les opérations pour chacun des objets de l'arbre sous le "répertoire virtuel"[39].

Nous ne sommes pas les premiers à reconnaître les mauvaises performances des connecteurs de stockage d'objets. D'autres ont essayé d'améliorer les performances, en sacrifiant l'exécution spéculative, puis en écrivant des objets directement à leurs noms définitifs, par exemple, DirectOutputCommitter[40] pour Amazon S3 ou en renommant la sortie Hadoop objets à leurs noms définitifs lorsque les tâches sont terminées (validation de la tâche) au lieu d'attendre le travail entier complète (engagement de travail) [41]. Cependant, en raison de l'inadéquation de l'impédance, ces tentatives ont conduit à de subtils échecs.

Les connecteurs actuels peuvent également entraîner des échecs et une exécution incorrecte, car l'opération de liste sur les conteneurs de stockage d'objets / seaux sont finalement cohérents. EMRFS [42] d'Amazon et S3mper [43] de Netflix surmontent la cohérence éventuelle en stockant des métadonnées de

[36] https://github.com/CODAIT/stocator
[37] https://aws.amazon.com/sdk-for-java/
[38] https://github.com/openstack/sahara-extra/tree/master/hadoop-swiftfs
[39] Les Object stores émulent les répertoires via une dénomination hiérarchique.
[40] https://github.com/apache/spark/pull/12229
[41] https://issues.apache.org/jira/browse/MAPREDUCE-6336
[42] https://aws.amazon.com/blogs/aws/emr-consistent-file-system/
[43] http://techblog.netflix.com/2014/01/s3mper-consistency-in-cloud.html

fichier dans DynamoDB[44], un fort supplément système de stockage cohérent séparé du magasin d'objets. Une fonctionnalité similaire appelée S3Guard[45] est en cours développé par la communauté open source Hadoop pour le connecteur S3a. Des solutions comme celles-ci, qui nécessitent plusieurs systèmes de stockage, sont complexes et peuvent introduire des problèmes de cohérence entre magasins. Ils ajoutent également des coûts puisque les utilisateurs doivent payer pour le stockage supplémentaire fortement cohérent.

Les nouveaux algorithmes de Stocator atteignent à la fois la haute performance et la tolérance aux pannes en prenant avantage de la sémantique de stockage d'objets. Cela réduit considérablement le nombre d'opérations sur le stockage d'objets ainsi que de permettre une approche beaucoup plus simple pour faire face à la sémantique finalement cohérente typique de stockage d'objets. Nous avons implémenté notre connecteur pour l'OpenStack Swift API[46] et le Amazon S3 API, et l'ont partagé en open source[47]. Nous avons comparé ses performances avec le S3aet Hadoop Swift sur une gamme de charges de travail et ont constaté qu'il exécute beaucoup moins d'opérations sur le magasin d'objets, dans certains cas aussi peu que le trentième des opérations. Depuis le prix d'un objet service de stockage comprend généralement des frais basés sur le nombre d'opérations exécutées, cette réduction le nombre d'opérations réduit les coûts en plus de réduire la charge sur le logiciel client. Il réduit également coûts et la charge pour le fournisseur de stockage d'objets, car il peut servir plus de clients avec le même montant de la puissance de traitement. Stocator augmente également considérablement les performances des charges de travail Spark sur le stockage d'objets, en particulier pour les charges de travail intensives en écriture, où il est jusqu'à 18 fois plus rapide. Stocator est en production dans IBM Cloud et a permis au projet SETI d'effectuer des calculs charges de travail Spark intensives sur des fichiers de signaux binaires de plusieurs téraoctets[48].

Le reste du **Chapter 6** est structuré comme suit. Dans la Section 6.1 nous présentons le contexteApache Spark et nous motiverons notre travail. Dans la Section 6.3 nous décrivons comment fonctionne Stocator. Dans la section 6.4 nous présentons la méthodologie pour notre évaluation de performance, y compris notre mise en place expérimentale et une description de nos charges de travail. Enfin, dans la section 6.5 nous présentons une évaluation détaillée de Stocator,comparant ses performances avec les connecteurs de stockage d'objets Hadoop existants, du point de vue de durée d'exécution, nombre d'opérations et utilisation des ressources.

A.5.1 Motivation

Pour motiver le besoin de Stocator, nous décrivons la séquence d'interactions entre Spark et son stockage système pour un programme qui exécute une seule tâche qui produit un seul objet de sortie comme indiqué sur la fig. A.3.

[44] https://aws.amazon.com/dynamodb/
[45] http://www.slideshare.net/hortonworks/s3guard-whats-in-your-consistency-model
[46] https://developer.openstack.org/api-ref/object-storage/
[47] https://github.com/SparkTC/stocator
[48] https://medium.com/ibm-watson-data-lab/simulating-e-t-e34f4fa7a4f0

```
val data = Array(1)
val distData = sc.parallelize(data)
val finalData = distData.coalesce(1)
finalData.saveAsTextFile("hdfs://res/data.txt")
```

Fig. A.3 Programme Spark qui exécute une seule tâche produisant un seul objet.

Table A.1 Ventilation des opérations REST par type pour le programme Spark qui crée un seul objet.

	HEAD Object	PUT Object	COPY Object	DELETE Object	GET Cont.	Total
Hadoop-Swift	25	7	3	8	5	48
S3a	71	5	2	4	35	117
Stocator	4	3	–	–	1	8

Au début d'un travail, le pilote Spark et l'exécuteur créent de manière récursive les répertoires pour le tâche temporaire, travail temporaire et sortie finale. Ensuite, la tâche génère le fichier temporaire de la tâche. À la tâche valider l'exécuteur répertorie le répertoire temporaire de la tâche et renomme le fichier qu'il trouve dans son travail temporaire prénom. Au moment de la validation du travail, le pilote répertorie de manière récursive les répertoires temporaires du travail et renomme le fichier trouve à ses noms définitifs. Enfin, le pilote écrit l'objet _SUCCESS.

Lorsque ce même programme Spark fonctionne avec les connecteurs Hadoop Swift ou S3a, ces opérations de fichiers sont traduits en opérations équivalentes sur les objets dans le magasin d'objets. Ces connecteurs utilisent PUT pour créer des objets octets zéro représentant les répertoires, après avoir d'abord utilisé HEAD pour vérifier si les objets pour les répertoires existent déjà. Lors de la liste du contenu d'un répertoire, ces connecteurs descendent le "Arborescence" listant chaque répertoire. Pour renommer des objets, ces connecteurs utilisent PUT ou COPY pour copier le objet à son nouveau nom, puis utilisez DELETE sur l'objet à l'ancien nom. Tout le répertoire zéro octet les objets doivent également être supprimés. Dans l'ensemble, le connecteur Hadoop Swift exécute 48 opérations REST et le connecteur S3a exécute 117 opérations. Le Table A.1 montre la répartition en fonction du type d'opération.

Stocator exploite la sémantique de stockage des objets pour remplacer le paradigme de renommer / renommer les fichiers temporaires. avantage de la dénomination hiérarchique pour éviter la création d'objets "répertoire". Pour le programme Spark en fig. 6.2 Stocator exécute seulement 8 opérations REST: 3 objets PUT, 4 objets HEAD et 1 conteneur GET.

A.5.2 Résumé

Au **Chapter 6**nous présentons un connecteur de stockage d'objets haute performance pour Apache Spark appelé Stocator,qui a été mis à la disposition de la communauté open source[49]. Stocator surmonte l'impédancenon-concordance entre les connecteurs open source précédents et leur stockage, en exploitant la sémantique de stockage des objets plutôt que d'essayer de traiter le stockage d'objets comme un système de fichiers. En particulier Stocator élimine le renommer paradigme sans sacrifier la tolérance aux pannes ou l'exécution spéculative. Il traite également correctement avec le Sémantique cohérente des magasins d'objets sans avoir à utiliser un stockage cohérent supplémentaire système. Enfin, Stocator exploite le codage de transfert par blocs HTTP pour diffuser les données telles qu'elles sont produites stockage d'objets, évitant ainsi d'écrire en premier dans le stockage local.

Nous avons comparé les performances de Stocator avec les connecteurs Hadoop Swift et S3a sur une gamme des charges de travail et a constaté qu'il exécute beaucoup moins d'opérations sur le stockage d'objets, dans certains cas aussi peu que un trentième. Cela réduit la charge à la fois pour le logiciel client et le service de stockage d'objets, ainsi que réduire les coûts pour le client. Stocator augmente également considérablement la performance des charges de travail Spark, écrire des charges de travail intensives, où il est jusqu'à 18 fois plus rapide que les alternatives.

A.6 Conclusions et Perspectives

Dans cette thèse, nous avons présenté des contributions à l'amélioration de l'efficacité des datacenters en termes de la réactivité du système. Des solutions ont été proposées pour atteindre différents objectifs:

- **Augmenter le niveau d'abstraction.** Nous avons défini, pour la première fois, une construction de haut niveau envoyé des applications analytiques, en se concentrant sur leur hétérogénéité, et leur cycle de vie de bout en bout. nous établi un nouveau problème d'ordonnancement, et proposé une heuristique flexible capable de gérer demandes hétérogènes, ainsi que diverses politiques de programmation, avec pour objectif ultime améliorer la réactivité du système sous des charges lourdes. Nous avons évalué notre politique d'ordonnancement en utilisant des traces de charge de travail réalistes et à grande échelle et montrent qu'il surpasse constamment l'approche de référence. Enfin, nous avons construit un système à part entière qui matérialise les idées des applications analytiques et leur calendrier. En utilisant notre nouvelle heuristique, nous avons pu réaliser des améliorations substantielles termes de la réactivité du système et de l'allocation des clusters.

- **Utilisation du cluster.** Nous avons présenté la conception du système pour un mécanisme dynamique d'allocation des ressources. nisme, qui peut généralement être appliqué aux cadres de gestion de cluster existants. Nous avons ciblé un famille spécifique d'ordonnanceurs d'applications analytiques, et matérialisé nos idées pour de tels ordonnanceurs. Nous avons introduit une

[49] https://github.com/SparkTC/stocator

nouvelle application de méthodologies d'apprentissage automatique à la fine pointe de prévision de l'utilisation des ressources, avec un traitement probabiliste qui permet de quantifier incertitude. Les informations de confiance ont été utilisées pour piloter les paramètres du système pics de demande inattendus de ressources. Nous avons réalisé une vaste campagne de simulation en utilisant des traces de production disponibles publiquement à partir des centres de données Google. Nous avons comparé notre approche à celui de Borg, et de discuter sur le compromis qu'une approche optimiste par rapport à une approche pessimiste la préemption de l'application implique. Enfin, nous avons présenté la conception d'un prototype de mise en œuvre de notre système, que nous utilisons dans un cluster académique au service des étudiants et des chercheurs. Nos résultats indiqué des améliorations substantielles en termes d'efficacité, qui se traduisent par un système capable de ingérer une charge de travail plus lourde avec le même nombre de machines.

- **Calcul et stockage désagrégation.** Nous avons réalisé une vaste campagne de mesure sur environnement de cloud computing privé impliquant la combinaison de plusieurs services d'analyse. Pour chaque scénario de déploiement, nous avons étudié les performances d'une variété de charges de travail d'application, y compris des applications intensives de lecture / écriture, de business intelligence et d'apprentissage automatique. nous présenté une notion intuitive de localité de données qui peut être utilisée comme un proxy pour classer différents services compositions, en termes de performance attendue. Nous avons examiné de manière critique la validité de notre intuition en fonction des charges de travail des applications, et identifier et expliquer les valeurs aberrantes. Nous avons montré preuve expérimentale de l'inadéquation de l'impédance entre le cadre informatique à grande échelle et deux couches de stockage importantes - les object stores et les block stores élastiques - et le mécanisme déduit à atténuer les effets négatifs sur la performance.

- **Stockage d'objets et incompatibilité des structures informatiques à grande échelle.** Nous avons présenté la conception d'un nouveau connecteur de stockage pour Hadoop et Spark qui exploite le stockage d'objets sémantique pour fournir de hautes performances et une exécution correcte face aux fautes et aux spéculations. Nous avons prouvé que cette solution fonctionne correctement malgré la sémantique d'objet finalement cohérente stockage, mais sans nécessiter de stockage supplémentaire fortement cohérent.

Avec le travail présenté dans cette thèse, nous avons réduit l'écart entre l'allocation des ressources et utilisation. Plus de travail peut être fait afin d'étudier de nouvelles techniques qui peuvent mieux modéliser les ressources utilisation à l'intérieur d'un cluster; de cette façon, nous pouvons encore réduire le nombre de défaillances dues aux ressources contention et réduire le gaspillage de ressources. Dans notre prose et nos expériences, nous avons toujours considéré que le processeur et la mémoire comme ressources principales, cependant d'autres types de ressources peuvent être pris en compte, comme le réseau bande passante et E / S disque. De plus, notre solution dynamique est à l'état de prototype. Fournisseurs publics doit garantir certains SLO et SLA à leur utilisateur. Plus de travail doit

être fait pour comprendre l'impact de notre approche pessimiste sur ces contraintes de niveau de service. Enfin, la méthodologie d'apprentissage automatique que nous avons proposé, peuvent échouer avec des applications interactives en raison de leur nature "humain dans la boucle". Il est difficile de prédire quand un utilisateur va interagir avec une application qui a été jugée inactive en regardant simplement l'utilisation des ressources. En même temps, il n'est pas facile de comprendre quand un l'application interactive est dans un état inactif; certaines ressources (par exemple, la mémoire) peuvent être occupés même lorsque les applications ne produisent aucun travail.