Scheduling the Spark Framework under the Mesos Resource Manager

Hans van den Bogert

Delft University of Technology

Scheduling the Spark Framework under the Mesos Resource Manager

Master's Thesis in Computer Science

Distributed Systems Group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Hans van den Bogert

24th August 2016

Author
Hans van den Bogert

Title
Scheduling the Spark Framework under the Mesos Resource Manager

MSc presentation
30th August, 2016

Graduation Committee

Prof. dr. ir. D. H. J. Epema	Delft University of Technology
Dr. ir. A. Iosup	Delft University of Technology
Dr. C. Hauff	Delft University of Technology

Abstract

Using clusters of servers and datacenters to process large numbers of data- and computation-intensive jobs is becoming mainstream. The need for large clusters is driven by the fact that many workloads are growing at a faster rate than the advances in single computer performance. To manage processing in a cluster of multiple computers, several frameworks have appeared over the years. These frameworks supply the user with convenient computation constructs and abstract from the low-level implementations to relieve the burden of inter-process communication and task placement in a cluster. However, these frameworks often assume total control over a static partition of the cluster, leading to under-utilization in times when one framework is over-committed with work, whereas another framework is idling. To overcome this under-utilization, and multiplex frameworks in clusters, cluster schedulers have been proposed, which sit on top of the hardware resources and schedule hardware resource leases to frameworks. It is not well described how these systems differ from each other. Furthermore, how to achieve *performance balance* between frameworks, such that multiple frameworks achieve similar performance metrics when running time-varying workloads is relatively unexplored.

We define a taxonomy which describes the combination of cluster schedulers and frameworks, called Two-Level schedulers in general. We characterize the multiple state-of-the-art cluster schedulers that are described in the literature or are used in practice. We distinguish multiple aspects which define the cluster schedulers. These aspects can dictate how frameworks interface with the cluster scheduler and can also influence framework performance.

We aim to achieve performance balance for multiple Spark frameworks running under the Mesos cluster scheduler. First, we evaluate the performance of a single framework running single interactive data analytics queries. We find multiple configuration parameters which influence the performance for interactive queries. However, we conclude that using Spark-Mesos for interactive queries results in either inefficient use of resources, or does not allow us to multiplex resources over multiple frameworks. We continue with achieving performance balance for multiple frameworks for non-interactive queries. We first establish a baseline performance balance, which we attain by using knowledge of the possibly different workload intensities run by the frameworks in a real cluster. Afterwards, we achieve similar performance balance, compared to the baseline, for up to three frameworks without knowing the workload intensity a priori. This is achieved by using a feedback loop controller, which updates resource share sizes allocated to frameworks dynamically, based on the online performance metrics of the frameworks.

“I hear and I forget.
I see and I remember.
I do and I understand.”
– Confucius

Preface

Computers have fascinated me since I was three years old typing memorized commands on an old TRS-80, just to start a game. The next best thing was using multiple computers in a network. In between the Bachelor and Master period, I've worked at a small data-center and became interested even more in networking and distributed computing in general. Thus, choosing a thesis at the distributed systems group doesn't come as a surprise. This project has at least taught me one thing: distributed systems are hard, and will probably remain interesting in the foreseeable future.

I would first like to thank my supervisor Prof.dr.ir Epema for his keen insights in the problems we've encountered and for helping to write a thesis. I would like to thank Bogdan Ghiţ and Aleksandra Kuzmanovska who helped by giving practical advise for the experiments in this thesis. I would also like to thank Ernst van der Hoeven, for providing access to his office. Niels Doekemijer, thanks for brainstorming along on countless occasions. I would also like to thank the alumni who have gone boldly before me, but who are so often forgotten in these prefaces. Rob Ruigrok, thank you for your obligatory *Radio Oranje* sessions on Friday afternoons. Steffan Norberhuis, well, thanks for being a Taylor Swift fan. I want to thank Elric Milon, Laurent Verweijen, Otto Visser, Maria Voinea, Wing Lung Ngai, and Stijn Heldens for making the thesis period a nice social experience as well. Ultimately, I want to thank my girlfriend and parents, whose support has been essential in bringing this thesis to completion.

Hans van den Bogert

Delft, The Netherlands
24th August 2016

Contents

Chapter 1

Introduction

Using clusters of servers and datacenters to process large numbers of data- and computation-intensive jobs is becoming mainstream. The need for large clusters is driven by the fact that many workloads are growing at a faster rate than the advances in single computer performance. This is not only due to Moore's law not applying anymore, but also the storage and memory performance of a single computer has not seen a strong enough increase over the years. To overcome this lack of sufficient single computer scaling, using multiple machines in a distributed manner, *seems* to be the obvious solution, though it brings problems not seen in single computer solutions, e.g., deployment and communication overhead.

To manage processing in a cluster of multiple computers, several frameworks have appeared over the years. These frameworks supply the user with convenient computation constructs and abstract from the low-level implementations to relieve the burden of inter-process communication and task placement in a cluster. A noteworthy framework is Hadoop. Hadoop is an open-source implementation of the MapReduce model, a model that has been popularized by Google [13], and that model has enabled the processing of jobs which span PetaBytes of data in a distributed manner. Other types of frameworks than job processing are Storm [30] for streaming applications, and OpenMPI for computationally intensive workloads.

The frameworks mentioned as well as others often assume total control over a pre-configured partition of the cluster, leading to under-utilization in times when one framework is over-committed with work, whereas another framework is idling. To overcome this under-utilization, and multiplex frameworks in clusters, several solutions have been proposed which all have in common that they sit on top of the hardware resources and schedule hardware resource leases to frameworks. These cluster resource schedulers positioned between the hardware and frameworks are called *global schedulers*. Examples are YARN [31], which was developed for multiplexing multiple Hadoop framework instances on a cluster, Google's Omega [26], which focuses on scalability for the scheduler itself, and Mesos [20], which acts as resource allocator, but still leaves many scheduling decisions, like task placement, to the framework.

A prominent combination of a computing framework and global scheduler is Spark [33], which is a relatively new framework in the same vein as Hadoop, running under Mesos. The Mesos global scheduler is a project initiated in 2009 by members of the UC Berkeley RAD Lab, and currently it lives on as a open-source product under the *Mesosphere* company. According

to Mesosphere, Mesos is being used by prime players in the industry, e.g., at Twitter spanning a cluster of 80,000 nodes [8], at Apple to run the Speech recognition service *Siri* spanning multiple thousands of nodes [4], as well as at other companies like Netflix, Ebay, Verizon, and Yelp.
Spark originates from UC Berkeley as well, and has also become an open-source product of a company called *Databricks*. It has been acclaimed for its more general computing model compared to MapReduce. Even with a more general model, Spark showed [9] that it was more efficient than Hadoop in a typical batch processing job, because it won the yearly benchmark for sorting 100 Terabytes of data in 23 minutes using 206 servers in the Amazon Cloud (EC2). Other work [27] comparing Spark and Hadoop also shows that Spark is faster for most types of jobs, which were previously only in the realm of Hadoop's capabilities. Besides batch processing, Spark has also gained a lot of popularity in data-analytics. In contrast to Hadoop, Spark allows for in-memory caching of data, which makes Spark faster in certain use-cases than what was traditionally possible. The speed-up due to caching can occur in the case that a query is run more than once. For subsequent query executions, the speedup for query run-times can be more than an order of magnitude, allowing for sub-second run-times, i.e., interactive querying. So, Spark is a framework which can handle workloads ranging from batch jobs to data-analytics queries, and Spark can even handle the interactive variant of the latter.

1.1 Multiplexing Multiple Frameworks

Using multiple frameworks in a cluster can be done in two ways. The first way is to statically allocate a group of nodes to a framework. The second way is using a global scheduler as already described above: The global scheduler allocates cluster resources to frameworks dynamically over time. In both the static and dynamic scenario, the framework schedules the placement of tasks to the resources it has at that moment. This means that in the case of the dynamic scenario, there are two types of schedulers at work, the schedulers in frameworks and the global schedulers. The combination of both types of schedulers are called Two-level Schedulers (TLSs).

TLSs provide us the mechanisms to run multiple frameworks on a cluster, without a framework requiring a static partition of the cluster. This gives flexibility for cluster administrators to provide for example, multiple instances of the same framework which can be beneficial for multiple reasons:

- Performance isolation, running multiple workloads on frameworks, such that the workloads do not interfere with each other in terms of achieved performance.
- Data isolation, end-users may want to use different instances of frameworks for different data-sets, for instance, for security purposes.
- Version isolation, running multiple versions of the same framework, can allow the end-users to gradually migrate to a new version of the framework, but keep current, tested, programs running on the old version.

When multiple instances of the same framework are running, it can be desirable that they achieve equivalent values of performance metrics for their workloads, even when the workloads have different intensities, i.e., the frameworks should achieve performance balance.

The paper introducing Mesos [20] shows that multiple frameworks can be run simultaneously and execute their workload faster than when those same frameworks are run in a cluster without Mesos, i.e., when each framework has a static partition of nodes in the cluster. The speedup is achieved due to statistical multiplexing: frameworks are allowed to grow beyond their initital partition of the cluster when other frameworks are not using (part of) their partition. However, the experiments and results in the paper do not show how similar frameworks which only differ in workload intensity can achieve a performance balance relative to each other.

1.2 Problem statement

The way global schedulers work, varies in different aspects, ranging from a monolithic omniscient scheduler to a distributed scheduler that is unaware of the actual allocations on the cluster. Previous work introducing new global schedulers [31, 20, 26] explain their systems using different aspects which are not necessarily comparable to each other. Therefore, in order to able to compare state-of-the art global schedulers, in this thesis we first try to answer the following research question:

RQ1 How can we characterize and differentiate existing two-level-scheduler designs?

Our original aim was to achieve a performance balance for interactive data-analytics queries executed on Spark frameworks under Mesos (RQ3 below). However during the experimentation phase of this thesis we had trouble making sense of the performance when running interactive analytics queries in a single Spark framework. Therefore, the first step is to evaluate the performance of a single framework and answer following research question:

RQ2 What is the performance of a single Spark framework under Mesos for interactive queries?

Our evaluation of Spark under Mesos for interactive queries will show to be problematic, because we have to use the so-called *fine-grained* mode of Spark to be able to perform performance balancing. However, fine-grained mode is inefficient for interactive querying. For these reasons, our conclusion is that with the current status of Spark and Mesos it is not possible to achieve meaningful performance balance. Therefore, we have only tried to achieve balancing for Spark under Mesos for non-interactive queries. This results in the last research question,

RQ3 Can we achieve a performance balance for non-interactive data-analytics queries between multiple Spark frameworks running under Mesos?

1.3 Approach

To answer the first research question (RQ1), we define a *two-level-scheduler* model and taxonomy which describes the combination of global schedulers and frameworks. We characterize the multiple state-of-the-art cluster schedulers that are described in the literature or are used in practice.

To answer the research questions RQ2, we use the combination of Spark under Mesos as our specific instance of a TLS. We will use this instance in our experiments on the DAS-4 [11], which is a university multi-cluster. We use an existing representative interactive query and measure the performance of this query for varying values of configuration parameters for both Mesos and Spark.

To answer RQ3, we first design three workloads consisting of non-interactive queries arriving at different arrival rates. For each workload we run an experiment and give two Spark frameworks the same workload. We verify if we achieve statistical multiplexing benefits for running the workloads and measure a baseline of the performance in terms of job slowdown.

Second, we try to achieve performance balance for two frameworks. We use experiments where the workloads have unequal arrival rates. We use multiple approaches ranging from default settings for mechanisms provided by Mesos, to altering parts of the scheduler in Mesos. Besides using mechanisms provided by Mesos we also introduce a feedback mechanism which uses online performance metrics from the frameworks and influences Mesos' resource allocation based on a policy. We report the performance balance by showing the difference of job slowdowns for the frameworks when running the workloads.

1.4 Thesis outline

The remainder of this thesis is structured as follows: In Chapter 2 we provide background information on two-level schedulers and frameworks and compare the former by using a proposed two-level scheduler model. We contribute a taxonomy by which existing two-level schedulers can be characterized and answers RQ1. In Chapter 3 we will evaluate the performance of a single Spark framework under Mesos for interactive queries for different values of configuration parameters. The results of the experiments answer RQ2 and show which parameters influence the performance. In Chapter 4 we define workloads with different arrival rates, and we evaluate how to achieve performance balance for multiple Spark frameworks. Our proposed system used in the experiments can be used as a basis for other work to achieve performance balance for data-analytics workloads. Chapter 5 concludes this thesis and provides directions for future work.

Chapter 2

Background and Characterization of Two-level Schedulers

Two-level cluster schedulers were designed to multiplex frameworks on a cluster. Adding a new type of framework is accomplished by designing a new framework which can interface with the underlying cluster scheduler, instead of altering the cluster scheduler itself, which has been the case for monolithic schedulers. Over the last few years, multiple types of these two-level schedulers have appeared. However, it is difficult to characterize the differences between these state-of-the-art Two-level Schedulers (TLSs).

This chapter describes the background to our work. In section Section 2.1 the field of Cluster scheduling is introduced and we contrast classical schedulers to two-level schedulers and we propose a model describing these cluster schedulers. Using this model, we describe existing two-level schedulers in Section 2.2 and answer RQ1: *How can we characterize and differentiate existing two-level-scheduler designs?* Examples of frameworks which can run in clusters by communicating with a cluster scheduler are given in Section 2.3. In Section 2.4 we explain Dominant Resource Fairness (DRF), which is the resource allocation policy used in Mesos. The information provided on DRF is needed to understand the results and discussions of Chapter 4.

2.1 Cluster scheduling

Commodity hardware in computer clusters as well as in data-centers have become mainstream in the last decade and scaling to bigger clusters is no longer matter of better hardware, but smarter software. On top of these clusters diverse frameworks for computation and data processing have been developed to ease the burden for developers and users to work with a cluster of nodes.

In this section we briefly describe cluster schedulers in Section 2.1.1, and we introduce TLSs in Section 2.1.2.

2.1.1 Classical job schedulers

In order to accommodate multiple users, i.e., to enable multi-tenancy, cluster schedulers have been introduced in many forms. These cluster schedulers often take jobs from end-users to be

processed on the cluster. It is the responsibility of the cluster scheduler to schedule a job's tasks according to the job's requirements as well as placing the tasks on appropriate cluster resources. Many of these cluster schedulers are available, e.g., HTCondor [29], Oracle Grid Engine [17]. The schedulers need knowledge of the type of job to successfully run it on the cluster. A difference in job types for example can be the property that its tasks can be run in any order (often called embarrassingly parallel), in contrast to jobs were a specific order, or communication between tasks is needed. An alternative to the classic cluster scheduler has recently become popular, called a two-level scheduler, which we will discuss hereafter.

2.1.2 Two-level schedulers

To accommodate multiple frameworks in a cluster, two-level schedulers have been devised. This type of schedulers abstracts away the allocation mechanisms of resources from the framework to a designated layer. The main reason for this abstraction is to allocate resources to multiple frameworks so they can co-exist on a cluster. Basically two-level schedulers provide a subset of the total cluster to a framework and leave the responsibility of task scheduling to the framework.

Motivation for the use of TLSs is given by first considering what the alternative might be. A commonly used framework is Hadoop [1], which is an implementation of the MapReduce programming model (further explained in Section 2.3.1). Although most computational frameworks are general, i.e., they allow many kinds of computation problems to be mapped onto the programming model of a specific framework, that does not imply that every solution to a problem can be computed *efficiently* using said framework. For example, mapping an iterative algorithm, where multiple iterations on input data are applied, onto Hadoop will be inefficient as MapReduce has no notion of iterations and will needlessly save intermediate data from every iteration to persistent storage, only to load it in the next iteration. Other reasons for having multiple frameworks running on a cluster exist, even for multiple instances of the same type of framework. For instance, it provides isolation in multiple ways:

- Performance isolation, running multiple workloads on frameworks, such that the workloads do not interfere with each other in terms of achieved performance.
- Data isolation, end-users may want to use different instances of frameworks for different data-sets, for instance, for security purposes.
- Version isolation, running multiple versions of the same framework, can allow the end-users to gradually migrate to a new version of the framework, but keep current, tested, programs running on the old version.

An obvious solution to having multiple frameworks running on a cluster is to partition the cluster's resources statically, i.e., to give parts of the resources to each framework. A schematic impression is given in Figure 2.1. Each square represents a resource, which can be a server node, a CPU core, or other resources depending on the context.

There are multiple inefficiencies with static partitioning, for one thing, high utilization of the cluster depends on whether every framework is using its complete partition. Taking Figure 2.1 as an example, even if framework 1 is using all of its resources, if framework 2 and 3 are

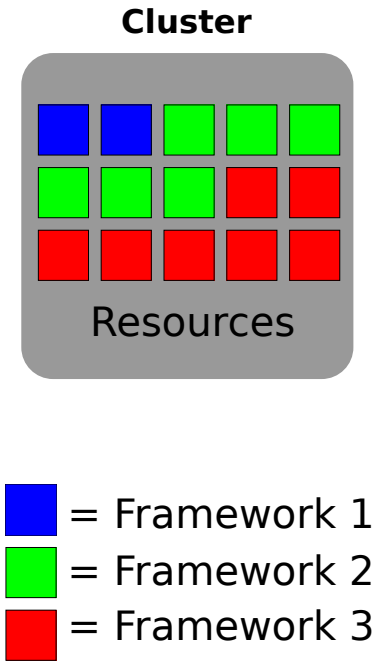


Figure 2.1: A schematic overview of a cluster with 15 resources statically partitioned among three frameworks.

idling, the cluster utilization is less than 13% (2/15). In order to achieve a higher utilization, a framework should be able to use another framework's resources if the latter is not using (part of) its resources.

2.1.3 Two-level Scheduler model

In the context of a compute cluster, or a data-center, resources are, e.g., CPU time, memory, disk space, network bandwidth. It is up to the resource scheduler to distribute these resources among contending frameworks.

The basic raison d'être of two-level resource scheduling in the context of cluster resources is to split a framework's computation model logic from its ownership of cluster resources. As an example, the Hadoop (version 1) framework, described in Section 2.3.1, assumes continuous ownership of cluster resources even when it is not processing jobs.

The *two levels* consist of:

1. a Global Scheduler (GS), which schedules the resources of the whole cluster to frameworks.
2. the framework schedulers who decide which outstanding tasks to execute on the resources they have acquired.

A schematic representation of the TLS model is given in Figure 2.2. In contrast to static partitioning, frameworks do not own the resources, instead frameworks acquire resources by means

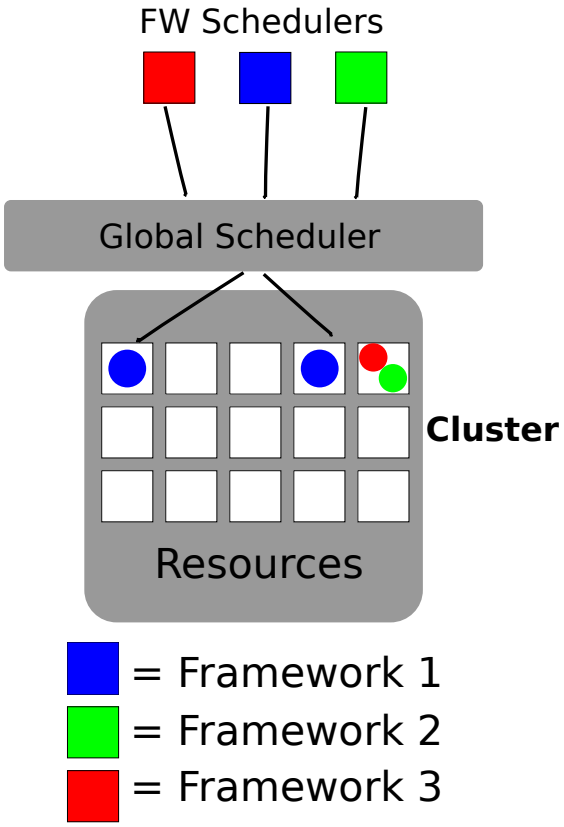


Figure 2.2: A schematic overview of a cluster with 15 resources scheduled by a global scheduler and three framework schedulers. In this instance, Framework 1 has acquired two resources. Framework 2 and 3 are both allocated in one resource unit.

of the GS.

We distinguish the following aspects in which the designs of GSs can differ from each other:

1. **Dynamic allocation** is the ability of the GS to make adjustments to the amount of allocated resources of a framework. On the other hand, with static allocation, the allocated resources are fixed and can't be adjusted during the lifetime of a framework. The ability to adjust allocations allows for better utilization of the resources, because resources coming available during an already started job can be used.

   A prerequisite for dynamic allocation to operate, is the co-operation of the framework. With static allocation, the opportunity of increasing utilization by adjusting allocation is infeasible.

2. **Pushed or pulled resources**, from the perspective of the GS, are two mechanisms used to initiate resource allocation. Pushing resources refers to the GS giving sources to frameworks, and vice versa for pulled resources. Pushing resources to frameworks allows the GS to set the pace of resource communication. In contrast, pulling resources, i.e., the GS

waits for incoming resource requests, can have as a result that many frameworks contend for resources at the same time, increasing load on the GS. The terms *pushed* and *pulled* will be used interchangeably with *offered* and *requested*, respectively, in the remainder of this report.

3. **Preemption** is the ability of the GS to revoke resources from frameworks. This aspect can be used by the GS to make resources available for frameworks which the GS deems more important. Preemption of jobs, or their tasks, can cause work to be lost. A distinction of two types of preemption can be made. With *co-operative* preemption, a framework is actively working on relinquishing resources, as requested by the GS. *Forced preemption*, is the GS revoking resources by forcefully stopping tasks running on those resources. It depends on the type of framework how well it can cope with these types of preemption.

4. **Granularity** refers to the scale of resource units being leased. For example, if a framework can request fractions of CPU, the granularity is fine-grained. If the acquiring of new resources is per unit of resources only (e.g., a whole node, or 10 nodes, etc), the granularity is coarse-grained. In Figure 2.2 an example of fine-grained resource allocation is shown for Framework 2 and 3. The two frameworks both have acquired a fraction of one resource unit. Having fine-grained resources can increase the utilization of the system under full load, as it is unrealistic that all workloads can be expressed in terms of coarse-grained units of resources.

5. **Distributed**, the topology of the GS . If there is a single component which communicates with frameworks, the GS is not distributed (e.g., Figure 2.2 shows just a single GS component). However, if a GS exists of multiple distributed components that can provide resources to frameworks, the GS is distributed. A distributed scheduler allows for a large variety of policies, i.e., there can be different schedulers for different types of frameworks. By having these specialized schedulers, scheduling time can be lower, because a scheduler does not have to take into account the different demands of multiple frameworks. However, the downside of concurrent schedulers is that eventually they have to schedule the same resources. Scheduling the same resources means that some form of communication is needed between the schedulers, to map tasks on resources, without multiple schedulers claiming the same resource.

6. **Global view**, from the perspective of a framework scheduler, does it have a global view of the cluster's state, or is its view restricted to acquired resources given by the GS. If a framework has knowledge of the cluster resources it can make better decisions where it wants its tasks to be placed. For instance, a framework can look at other tasks which are already running on a node, and decide if it wants to allocate its tasks on that host or not, as other tasks might interfere with other tasks in terms of performance.

7. **Scheduling separation** is a ordinal measurement of how separated the responsibility of resource and task scheduling is between the GS and frameworks. In one extreme we have a central scheduler which handles both task scheduling, and scheduling tasks on that resource. In another extreme we have multiple frameworks which decide where to place

Aspect	Mesos	YARN	Omega	Sparrow	Quasar	Koala-F
Dynamic Allocation	+	+	+	+	+	+
Pushed resource	+	-	-	-	+/-	+/-
Preemption	-	+	+	-	+	+
Granularity (Fine)	+	+	-/?	+	+	-
Distributed	-	-	+	+	-	-
Global view	-	-	+	+	-	-

Table 2.1: Scheduler aspects of the discussed Two-level Schedulers

their tasks in a cooperative manner without a GS. In between these extremes we have a separation of the resource scheduling, handled by a GS, and the task scheduling, handled by frameworks.

In the next section we will discuss a number of GSs using the aspects defined above.

2.2 State of the art Global Schedulers

In this section we describe the six most important GSs using the above discussed aspects. In Table 2.1 we give an overview of the frameworks using the above described aspects.

The *scheduling Separation* aspect is depicted as a spectrum in Figure 2.3. The left extreme of

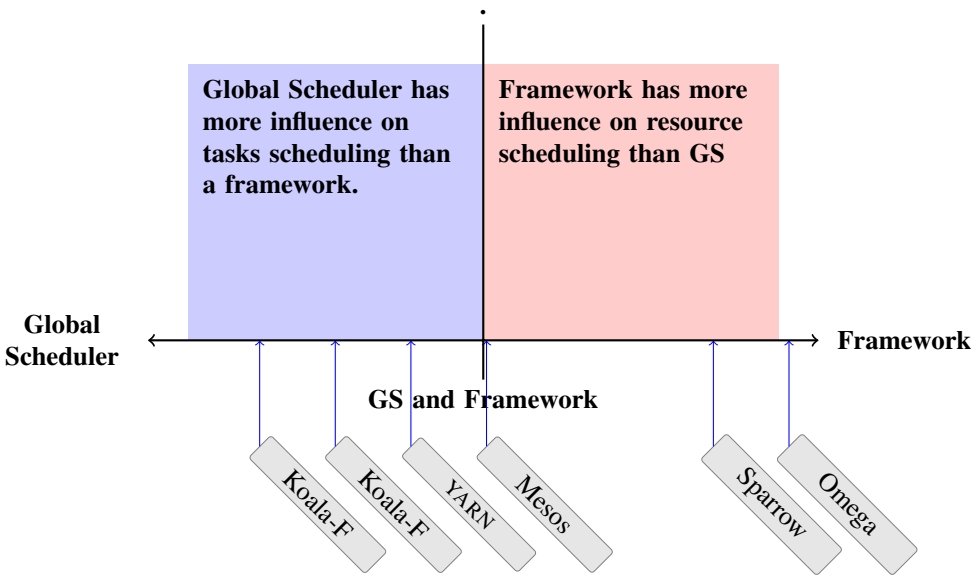


Figure 2.3: Separation of resource and task scheduling between Global Scheduler and Framework.

the spectrum is a TLS where resource allocation and task allocation is handled by the GS. The classical cluster schedulers fall in to this category (Section 2.1.1). In the middle of the spectrum is a clear separation between resource and and task scheduling by the GS and frameworks, respectively. Mesos is a prime example of this. In the right extreme we have frameworks which handle their own resource scheduling and task scheduling. In the following subsections we will handle each GS shown in Table 2.1.

2.2.1 Mesos

The Mesos project originates from UC Berkeley's Amplab as a research project to manage a cluster's resources to provide resources to application frameworks. The project is currently incubated under the *Apache Software Foundation*.

Mesos comprises a Mesos master process, and a Mesos slave process for every worker node. The slaves send information about the amount of processors and memory (and disks) to the master, so that the master knows their state. Frameworks in need of resources register at the master in order to be a candidate for a resource offer given out by the master. A resource offer is a tuple of CPUs and memory owned by one slave, available to the framework.

Mesos starts a new round of resource offers every *allocation interval*, which is one second by default. At every round Mesos uses an allocation algorithm called *Dominant Resource Fairness (DRF)* [19] to determine how many resources every framework should be allowed to use (explained in finer detail in Section 2.4). Once a framework gets an offer it has to decide what to do with it, it can either decline if the resources do not suit its needs, or use (part of) the resources to do its work. When it chooses the latter it sends back a description of the tasks to be run, where each task consists of three parts:

1. the fraction of resources to take with a maximum of the total resources in the offer. Any fraction of a CPU and any amount of memory can be taken up for offer.
2. a description of a framework specific *executor*, which is a main process per node, which can accept task descriptions.
3. a task description. The description depends on the framework, and is forwarded by the Mesos master to the executor running on a slave node.

The master, on receiving the description, starts the executor and after success provides task descriptions to the executor. A typical interaction between a framework and Mesos is shown in Figure 2.4. In the scenario of the figure the slaves and frameworks have already been registered. The sequence of messages is denoted by numbers on the directed edges. The messages show a complete round of offering resources to a framework,

1. Slave 1 tells the Mesos master its current free resources
2. The Mesos master offers Framework 1 the resources of Slave 1
3. Slave 1 accepts part of the offer, and sends a message with 2 task descriptions. The tasks do not use all resources that Framework 1 was offered; implicitly Framework 1 rejects 1 CPU and 1 GB of memory.

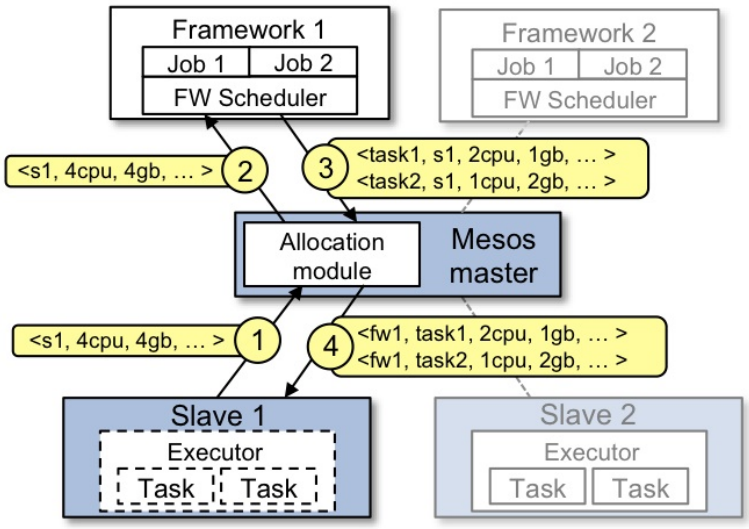


Figure 2.4: An example of an accepted resource offer in Mesos. The numbers in the circles represent the chronological ordering of the messages. Taken from: [20]

4. The master forwards the task descriptions to Slave 1.

In order for Mesos to give resources to a framework, the latter has to support the interfaces of Mesos, so frameworks that have not been developed to be run on Mesos initially, will need to have parts of their code rewritten.

Also note that, in contrast to what was explained the previous section, Mesos currently does not handle offers in the (CPU-) time dimension but instead allocates a fraction of a CPU indefinitely, i.e., it does not implement preemption to restrict a framework's lease time of resources.

Mesos is characterized by the following aspects:

1. Dynamic allocation, since frameworks are able to accept new offers continuously, and actually have to, since the lifetime of offers ends when a task is done.

2. Pushed based, since fine-grained resource offers are made by a single master, However, with the current state of Mesos once a framework accepts and starts a task, resources can not be revoked. Furthermore, frameworks can only be aware of the resources given to them by the master, prohibiting the frameworks from making decision which might need global knowledge for task placement.

2.2.2 YARN

Similar to Mesos, YARN [31] can accommodate multiple frameworks, and its GS is pull based, which means that frameworks signal to the GS that they need resources.

YARN's reason of existence is born from the desire to better utilize Hadoop workloads. Hadoop is tightly coupled with its own scheduler. Prominent design goals for YARN are allowing support for different programming models (so not restricted to MapReduce workloads) and a

more flexible resource model. Hadoop (version 1) runs MapReduce workloads on worker nodes which have a fixed number of map and reduce slots. However, the most efficient ratio of map and reduce slots depends on the application, but the number of slots are set once the framework is started. In Hadoop (version 2), which uses YARN, MapReduce frameworks are started on the fly for every job. At start-up, the job submitter has flexibility for setting its own ratio for map and reduce slots, which may result in more efficient usage of resources.

The YARN model consists of a single *Resource Manager* which functions as the GS in our general model. The Resource Manager orchestrates worker nodes by interfacing with a per-node *NodeManager*. If an end-user wants to use the cluster it must create a job which describes a YARN *Application Master*, which for all intent and purposes can be considered a framework. An application master thereafter communicates to the Resource Manager and can ask for more resources to place its tasks on, granted resources are called containers in YARN nomenclature. An architectural overview of YARN is given in Figure 2.5 where two clients, i.e., frameworks, schedule containers. The figure includes the steps needed (indicated by numbers) for scheduling and executing a job in YARN. Descriptions for the steps are as follows:

1. A client submits a job description to YARN's global scheduler, the Resource Manager. The job description includes information on how to boot the Application Master.
2. The Resource Manager looks at the job description and schedules resources, in the form of a container. The container is then instructed to start the Application Master.
3. The Application Master makes itself known to the Resource manager. Now that there is a registration of the Application Master in the Resource Manager, the client can access the Application Master.
4. If the Application needs more resources to run its tasks on, it can negotiate with the Resource Manager to acquire them
5. When the Application Master has gotten resource allocations from the Resource Manager, it launches container, by means of submitting a container description to a NodeManager.
6. Once containers are running they can report their status back to the Application Master.
7. During the execution of the job's work, the client can connect to the Application Master to monitor the application, e.g., get job progress. However, this functionality depends on the Application Master.
8. If a job finishes execution, the Application Master unregisters itself from the Resource manager. At this point all resources, i.e., containers the Application Master started to execute tasks, and the container of the Application Master itself; have now been deallocated.

Strictly speaking, YARN is not a TLS. In contrast to a Mesos framework, a YARN Application Master does not schedule, but merely requests the Resource Manager for extra containers. This might seem like a semantic quibble between the push-based (Mesos) vs pull-based (YARN) aspect. However, there is an architectural difference between the two, as Mesos allows a framework to schedule a task on a worker node, but YARN only allows an Application Master to describe a task, after which the GS decides on which worker node to put it.

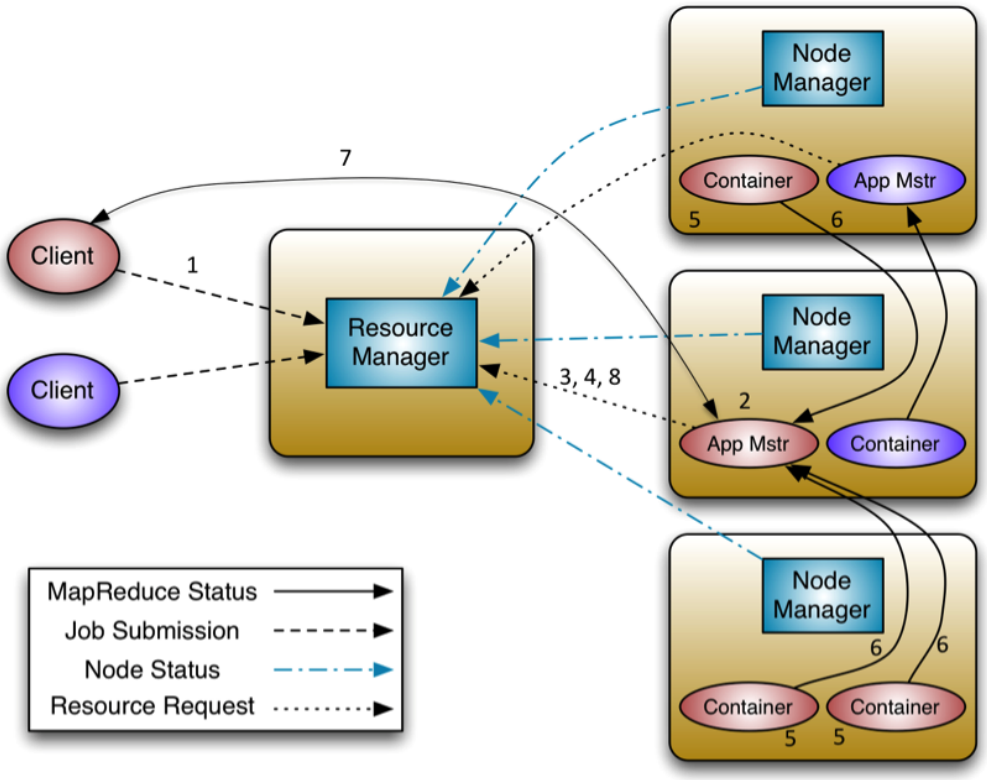


Figure 2.5: An example of resource negotiation in YARN. Two clients start application masters (MapReduce instances), which in turn successfully request containers to place their tasks on. Taken from: [2]

2.2.3 Omega

Google's Omega is different than the aforementioned schedulers, which from an architectural point of view are centralized schedulers, i.e., there is an entity which holds all resource allocation information. In the case of Omega, many GSs can exist, which have a shared data-structure representing the cluster's state in a best-effort manner.

Although the authors [26] contrast Omega to two-level schedulers, we consider Omega as a specialization of a two-level scheduler. The GSs also function as frameworks, i.e., they will schedule tasks onto resources. spectrum in Figure 2.3. As every framework only has a best-effort state of the system, two frameworks can try to place their tasks on the same resources if one of the frameworks' state of the cluster was stale. In that case a conflict resolution mechanism is needed to solve the problem of which framework gets the resources. The conflict resolution is not discussed in detail in the original paper, but several points of interest are: *1*) higher priority tasks

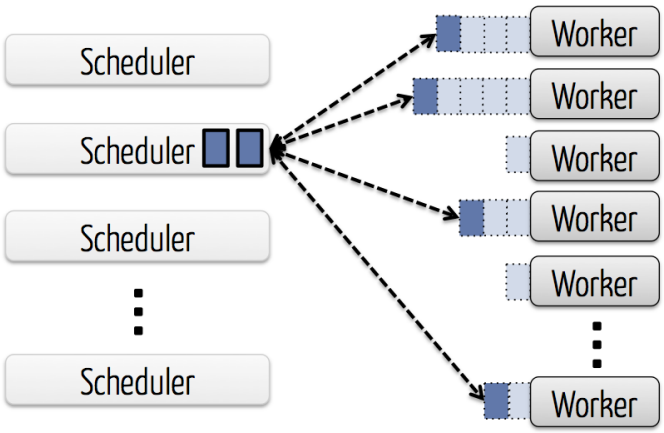


Figure 2.6: An example of the *two-choices load-balancing* technique used by Sparrow to place tasks on worker nodes. There is one job present in a scheduler which consists of two tasks. Four probes (edges) are made to random workers. Taken from: [10]

can preempt lower ones and *2*) task allocation tends to be incremental for Google workloads, i.e., if not all tasks are allocated due to conflicts, proceed to start the allocations that did succeed and retry allocations for the conflicting tasks

The responsibility of scheduling resources among the frameworks is distributed among the frameworks. Co-operation is needed during the conflict resolution, which makes the approach of Omega. This is possible for Google, because their clusters only host frameworks which are created by Google. However in environments where there are multiple tenants, it can be a problem to let multiple frameworks work together co-operatively when resources are limited, as there is no incentive for tenants and their frameworks to do so.

Omega is placed on the right extreme of the *two-level scheduling separation* in Figure 2.3, since both resource scheduling and task scheduling are the responsibility of the framework. In this sense, the model of Omega can be seen as the opposite of a classical scheduler where all scheduling logic is placed in a single global scheduler.

2.2.4 Sparrow

Glssparrow [23] is architecturally different from Mesos, YARN in the sense that it does not require an up-to-date state of the job allocations in the cluster, and Sparrow also does not *try* to have an up-to-date state, which is in contrast to Omega. Instead, a Sparrow cluster exists of multiple framework schedulers and worker nodes. Frameworks place tasks according to the *power of two choices load balancing technique* [22]. The load balancing technique is shown in Figure 2.6 and works as follows:

1. A job with m tasks is submitted to a scheduler. (One job with two tasks are shown in Figure 2.6)

2. The scheduler creates $2 \cdot m$ probes which it sends to random worker nodes in the cluster

3. The probes are placed in the queue of the worker. (Shown as directed edges in the figure)

4. When a probe reaches the head of queue, the worker informs the scheduler which placed the probe with a signal indicating the task can begin.

5. On receiving the signal, the scheduler responds by acknowledging the start of the task, or it cancels the task if the task has already started on another worker.

One of Sparrow's main design goals was to be able to provide low latency when scheduling. To accomplish this goal the Sparrow architecture expects long running frameworks to be already available at time of job submission, i.e., an end-user submits to a framework, hereby skipping framework start-up time.

Ousterhout et al. claim [23] that a decentralized scheduler has attractive properties, e.g., by scaling horizontally in scheduler servers, in this regard it resembles Omega. Their experiments show that the median response times of jobs are within 12% of an ideal scheduler. The design of Sparrow also has some assumptions which might not always hold. Sparrow assumes many tasks with homogeneous runtimes. If the task runtimes are heterogeneous and the total system is under high load, then small tasks will have a high chance of having to wait for large tasks in front of them in the queue on the worker node. P. Delgado et al. [15, 14] have proposed a hybrid scheduler which handles small and large tasks separately, by using a model like Sparrow's for small tasks and a centralized scheduler for long tasks.

Although Sparrow is distributed by design and has no actual GS entity, we did not place it on the right extreme of our TLS separation spectrum. The frameworks can decide to place tasks on resources to a large extent, but it is random to some extent as well.

2.2.5 Quasar

Previous GSs let job submitters, or their delegate frameworks, choose the amount of resources a job is going to need, i.e., a job has a strict resource request. Quasar [16] does not use job accompanied reservation information, but takes into account a job's performance constraints. For example, in the case of the Hadoop framework, a performance constraint can be a maximum execution time for a specific job. Another performance constraint can be the number of queries per second when low-latency is required for frameworks like an interactive Spark instance.

The decision how many resources a job needs, shifts from the job submitter to Quasar. In order to decide how many resources a job needs, Quasar does the following:

1. It profiles part of the workload, by running it on nodes.

2. The profiling information from step 1. is combined with existing offline profiling of node characteristics under different circumstances. A classifier takes the profiling information to estimate the needed fine-grained resources to meet the job's performance constraints.

3. With the resource estimate, Quasar then runs the job by setting up the framework and continues to monitor the job's performance. If at any point resources are idling or the constraints are not met, Quasar deallocates or allocates more resources, respectively.

If Quasar tries to scale up resources, but not enough resources are available, low priority jobs are preempted. The low-priority jobs' deallocated resources are used to scale up the job which is not meeting its performance constraints.

2.2.6 Koala-F

Koala-F is a research project of A. Kuzmanovska et al. [21]. It is an extension to the KOALA scheduler created at the University of Delft. KOALA started as a more classical job scheduler and has had many additions since, e.g., multi-cluster and cloud awareness, on-demand Hadoop cluster deployments, etc. Koala-F adds the functionality to schedule for frameworks instead of jobs.

Koala-F accepts jobs, which describe a complete framework, and it sets up the framework with an initial amount of resources. The initial size is set by the cluster administrator and can differ for each job. Multiple framework-jobs can be accepted by Koala-F until all resources are partitioned among frameworks. Koala-F and its frameworks use a feedback mechanism where frameworks report their performance to Koala-F at a certain rate. The performance is expressed on a scale of three values, each identified by a color: Green, Yellow, and Red. Green indicates that a framework performs well and may even perform well enough if it had less resources. Yellow indicates that a framework performs well enough, and does not have resources to spare. Red indicates that the framework is underperforming. These colors are defined differently for each framework, and two threshold values for a framework-specific metric need to be profiled beforehand to provide a mapping of the metric space to the three colors. In the case of idle resources in the cluster, frameworks which are performing under the Red mode, will be given resources from the idle pool. Frameworks performing in the green mode will shortened on their resources, and the resources are marked free.

If there are frameworks of the color yellow and red running concurrently, and there are no idle resources (this implies there aren't frameworks running in the green mode), then resources from a yellow framework are given to a red framework. The frameworks run until their workload are done, or there aren't any job in their queue. In these cases, Koala-F removes the framework from the system and the resources are marked as idle.

The resource allocation is relatively coarse-grained compared to other GSs, as Koala-F gives resources as number of nodes. Koala-F is placed on the left of the spectrum of the separation of scheduling, because Koala-F dictates which resources are given to frameworks. This means that it is difficult for frameworks to have a view of the cluster resources, thus making it more difficult to place tasks on appropriate resource.

2.3 Frameworks

Cluster frameworks provide an abstraction over the cluster resources such that developers can write applications according to the framework's computation model. This relieves the developers to explicitily handle, e.g., inter-process communication, task placement, etc. There exist numerous cluster frameworks with varying abstractions and computation models, Message Passing

Interface (MPI) for computation and communication intensive applications, Hadoop for batch processing using the MapReduce model, Spark for multiple types of jobs, etc.

In the next two subsections we will discuss Hadoop and Spark. The model of Hadoop, *MapReduce*, is widely used in the industry and is provided as background knowledge. Spark is a framework gaining popularity, and is used in the experiments of Chapters 3 and 4 and its computation model is assumed to be known to the reader in those chapters.

2.3.1 Hadoop

Hadoop is a framework which has been in development since 2002 and was originally developed by Yahoo. Yahoo invested time in a computation platform was to handle the large amount of data acquired through web crawling in order to index websites pages. Since 2002 the then named Nutch project took design ideas from Google's papers on a distributed filesystem [18] and MapReduce [13]. MapReduce is a computation model which scales well over multiple nodes.

An Hadoop cluster exists of a single[1] master and multiple worker nodes. Hadoop has two services running on the cluster. The first service is the data layer, called Hadoop Distributed File System (HDFS) which is responsible for distributing data safely on multiple nodes. Data safety is achieved by replicating blocks of data to different nodes. HDFS consists of two types of processes: a *NameNode* installed on the master node, which acts as an index to all data in the cluster. The actual data is distributed over the worker nodes which run a *DataNode* process, which can serve and accept data from/to disk.

The second service is Hadoop's implementation of MapReduce which we will refer to as MapReduce hereafter. MapReduce also exists of two types of processes: A *JobTracker*, which is located on the master node and accepts jobs from end-users. The JobTracker translates a job to multiple tasks and sends those to *TaskTrackers*, which are on all the worker nodes. The first types of tasks to be run represent the Map operation.

A MapReduce job describes two functions and are (not suprisingly) called the map and reduce function. The end-user gives a specific implementation of these two functions which are typically executed on data located on HDFS. The JobTracker translates a single job to multiple tasks of map and reduce type. Once the task is accepted by a TaskTracker, a single map task gets data from the DataNode typically as a list of items and *maps* over these items with the specific transformation the end-user implemented. The output is again a list, of transformed items as key-value pairs. At this point intermediate output is scattered on all the worker nodes. The reducer tasks are also run on the worker nodes and Hadoop provides the intermediate output from the Map tasks as input to the reducer functions. The manner in which Hadoop provides intermediate data to a reduce task is in such a way that each reduce task gets a partition of the total intermediate data sorted by key. In order for a sorted partition to be located on one node, multiple unsorted records of intermediate data, need to be transferred to a particular node. Once a data partition is gathered from other worker nodes, the reduce task on that worker node can begin processing. This gathering of data from and to worker nodes, is called the *shuffle* phase.

[1]There are exceptions where there are multiple master nodes in the cluster, for scaling up or adding redundancy to the master node functionality

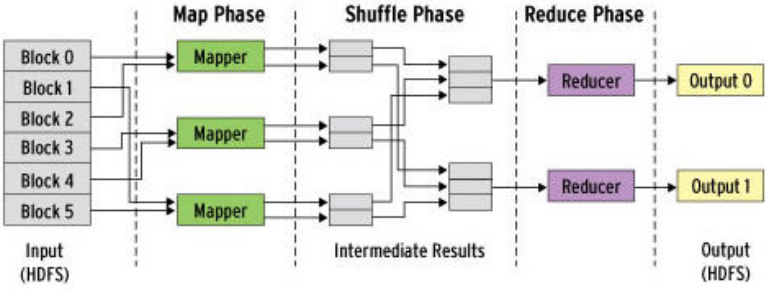


Figure 2.7: A MapReduce job broken down in three phases. Taken from: [7]

Each reducer runs the reducer function for every unique key and its corresponding values, and it reduces it to zero or one output record. All the records for one reducer task are stored as output in HDFS. A schematic overview is given in Figure 2.7. The figure shows the three phases of a MapReduce job as we've discussed above.

Hadoop has an important property of trying to place Map tasks on machines which hold the data locally on disk. This is done so that Map tasks do not have to transform data which is stored on another node. This saves implicit copying data from the node holding the data, to the node executing the map task. This principle of sending tasks to the data, is called *Data locality*

Not all problems which can be expressed as a MapReduce jobs run efficiently in Hadoop. An example of a type of problem which does not fit Hadoop's Mapreduce is an iterative algorithm. An iterative algorithm can consist of multiple map and/or reduce phases. However, MapReduce only supports one Map and one Reduce phase. After the Reduce phase the output is written to HDFS. If an iterative algorithm is written as multiple MapReduce jobs, then data is needlessly written and read to and from HDFS between jobs. The Spark framework, discussed hereafter, does support iterative programs.

2.3.2 Spark

At the framework level, we evaluate Spark [33], which was initially developed as an alternative to Hadoop to better fit iterative jobs. Spark's advantage comes from its ability to cache results in memory, which might be needed in successive phases of an iterative job. Spark's highest level abstraction is an application which accepts job submissions and further orchestrates communication to the worker nodes, called *executors*, through a *driver* sub-process. A schematic overview is given in Figure 2.8

On job execution, first the input data is loaded from persistent storage by the executors and partitioned into multiple data sets which are evenly distributed among worker nodes. Depending on the input size, there may be tens of these partitions per worker node. This distribution of data into memory is defined as a Resilient Distributed Dataset (RDD) [32] and is shown, schematically, in Figure 2.9a. Second, when the input data is loaded, a Directed Acyclic Graph (DAG) of tasks is created from the high-level Spark transformations (like *map, filter, reduce*) described in the computation part of the job. The vertices and edges in the DAG represent RDD partitions and transformations respectively. Thus, in Figure 2.9b, the left-most vertices represent

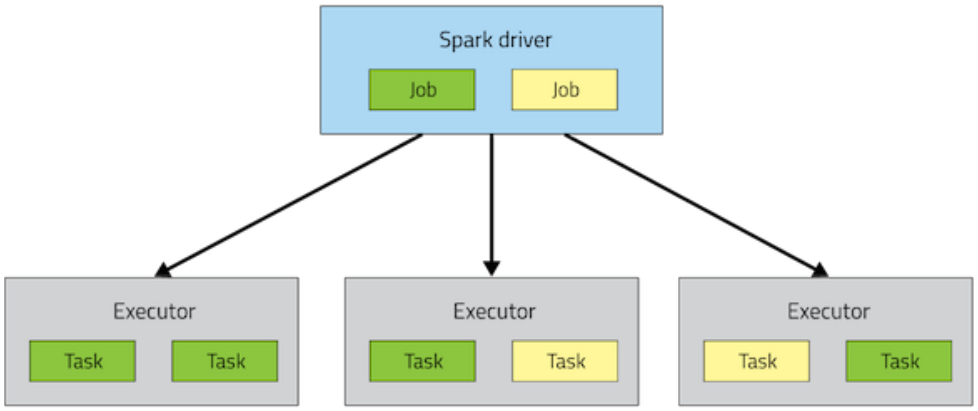


Figure 2.8: A Spark application, existing of a driver process which communicates to executors, which are the worker nodes in a Spark cluster. This particular application places two jobs on the executors. Taken from: [3]

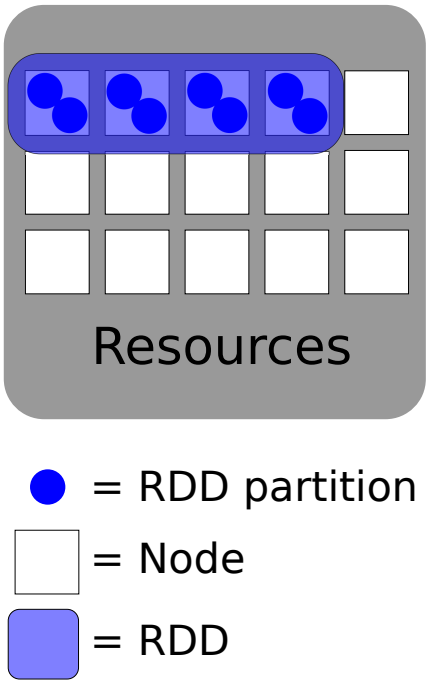
the input data and any other vertices represent partitions from new RDDs as a result of applied transformations.

The DAG also shows the transitive dependence between vertices, if they are connected by an edge. For example, in Figure 2.9b, the partition in RDD3 depends on all partitions in RDD2, and transitively, also the partitions in RDD1. In order to execute the complete example job, two stages are needed. One stage will execute the map function on the input data, RDD1, in parallel. Stage 2 takes the output of the previously applied map functions, RDD2, and applies the reduction function in parallel. The output of the reduction in this example outputs a single partition as RDD3.

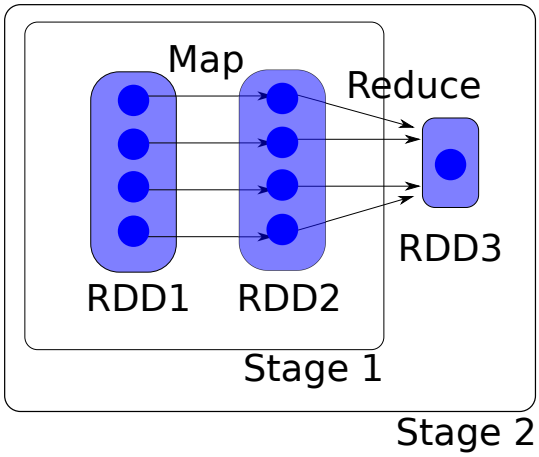
For every transformation on a partition, Spark creates a task. These tasks are mapped to CPU cores on worker nodes. For example, if we have two CPU cores in total and apply the map functions on four partitions as depicted in stage 1 of Figure 2.9b, it would take two *waves* of two tasks to complete the first stage.

As the name implies, RDDs also provide fault tolerance by offering resilience if a worker node fails. In order to achieve this, all nodes log the transformations they have applied previously. When a node (N_L) is deemed lost, other nodes each reload a fair share of input data which belonged to N_L and replay all the transformations which have been performed previously. By using this replay construct, performance penalty is paid only when a node has failed. Other solutions to fault-tolerance are often in the form of data replication. However using data replication incurs a continuous penalty in the form of extra data transfer between nodes, even when nodes are not failing.

Spark has different back-ends to run its executors on. At the time of writing, it supports its own specific back-end, called *standalone mode*, Mesos, and YARN. Specifically for Mesos, Spark implements two variants, fine-grained and coarse-grained mode. In fine-grained mode, Spark waits for an offer from Mesos for every outstanding task it has, i.e., it maps one Spark task to one Mesos task. In coarse-grained mode, at the start of a Spark application, all available



(a) Distribution of RDD partitions over nodes



(b) An example DAG, composed of two stages. Edges represent transformations on RDD partitions.

Figure 2.9: Two ways of looking at Resilient Distributed Datasets, (a) shows a RDD distributed over multiple nodes. (b) shows an DAG of an example job.

Mesos slaves will run the Spark executor without any interference from Mesos, and the Spark driver talks to these executors directly after they've been started. In contrast to fine-grained mode, in coarse-grained mode Mesos resources are allocated for the duration of the application. The benefit is that no communication, besides the initial setup of the executors, goes through the Mesos master, which yields better response times for small jobs. In fine-grained mode small jobs may incur a lot of overhead as communication costs are not amortized by their computation time. On the other hand, coarse-grained is not strictly better as it doesn't allow for cluster resources to be shared by other frameworks, as cluster resources are hold onto longer, giving no opportunity for other frameworks to get resources.

SparkSQL Spark has a submodule for working with data-sets which have the property of being structured into fixed order. A simple example of a structured data-set is a comma delimited file, where each row consists of multiple records, and the order and data type of those records is consistent within the file. The Structered Query Language (SQL) has been developed to query such data-sets, and SparkSQL is Spark's implementation for an SQL interface to query distributed data-sets with Spark as the execution engine.

To be able to use SparkSQL on structured data-sets, Spark has to understand the schema, i.e., the order and types of the columns, which make up the data-set. If an end-user provides a

schema, Spark can translate SQL queries to Spark primitives like the previously mentioned *map*, *filter*, and *reduce*. This allows end-users who do not have extensive knowledge of Spark, but who do know SQL, to be able to query large data-sets.

2.4 Dominant Resource Fairness

The DRF allocation policy used in Mesos has already been mentioned when describing Mesos (Section 2.2.1). We will elaborate on it further, because the mechanism and policies used by DRF are important in the interpretation of the results in Chapters 3 and 4. The role of DRF is to ensure that each framework receives a fair share for *multiple* types of resources. Types of resources can be CPUs, memory, etc, and the resource demands from frameworks are the input for DRF

Before we can explain the policy of DRF we explain the mechanisms and notions used in DRF. There is a notion of a *dominant resource* for a framework, that is, the resource that is most in demand by that framework. To be able to compare different kinds of resource demands, each type of resource is translated to a percentage of the total of that resource in the cluster. DRF equalizes the dominant resources for each framework.

As an example, let us take two frameworks in a system. The total resources configured in this system are 9 CPUs and 18 GigaByte (GB). This example is also shown graphically in Figure 2.10. Framework A has outstanding tasks, each task requires one CPU and and four GBs of memory. Therefore, Framework A's dominant resource is memory since each task would require an allocation $1/9$ of the total CPUs, compared to $2/9$ of the total GB in the system. Framework B's tasks require three CPUs and one GB of memory. Following the same reasoning as for Framework A, B's dominant resource is CPU since it demands $1/3$ of all CPUs per task and only $1/18$ of total memory.

The total amount of resources allocated to a specific framework by DRF is based on its dominant resource. DRF will give each framework equal dominant resource shares. Note that if the dominant resource type differs between frameworks, the equal share can be greater than $1/2$. For instance, for framework A and B using DRF shows that the frameworks can run three and two tasks concurrently, respectively, resulting in an equal dominant resource share of $2/3$. Note that any other allocation would be unfair, since that would mean the frameworks do not have an equal dominant share of the cluster. Mathematically, the solution of DRF with two resource types can be found by expressing it as a linear optimization problem. Let a and b represent the amount of tasks allocated to Framework A and B respectively. Then Framework A will receive $(a \text{ CPU}, 4a \text{ GB})$ and Framework B will receive $(3b \text{ CPU}, b \text{ GB})$. The restrictions for the optimization are twofold. One, the total amount of CPUs and memory allocated, should not exceed the cluster's total. Two, the dominant resources of both Framework A and B should be equal. The above can be written in standard form as follows:

$$\begin{array}{rlcll} \max & a, b & & & \\ \text{s.t.} & a + 3b & \leq & 9 & \text{represents cpu demands} \\ & 4a + b & \leq & 18 & \text{represents memory demands} \\ & \frac{2}{9}a & = & \frac{1}{3}b & \text{represents dominant resource shares} \end{array} \tag{2.1}$$

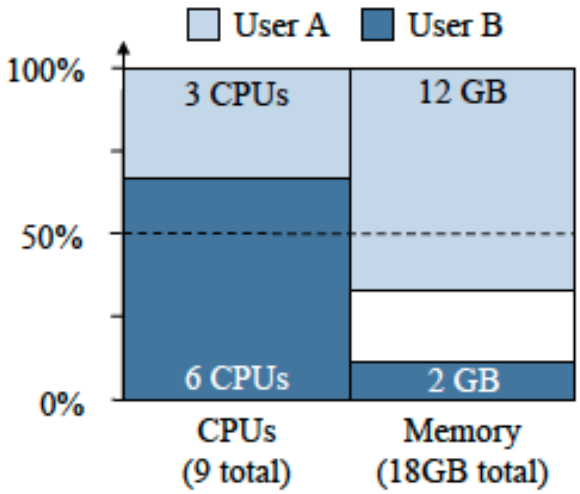


Figure 2.10: An example of DRF with a cluster consisting of 9 CPUs and 18 GB of memory. Two frameworks with different types of tasks are allocated their fair share of the cluster, according to DRF: Each has $2/3$ ($\approx \%66$ in figure) of their respective dominant resource.

Up to this point we explained DRF for a specific instance in time. However, in a running Mesos system, a framework's demands may change in time. For instance, specific tasks finish, and as a consequence the resources which were used by these tasks are now unallocated. Having unallocated resources also means that it's possible that a framework is not getting its equal resource share. To accommodate online, continuous resource scheduling, Mesos invokes DRF every period, called the *allocation interval*. However, Mesos does not solve the above optimization problem, but uses an approximation to the problem. During each interval the following operations are executed:

1. For every Mesos slave which has unallocated resources, make a resource offer which contains all the unallocated resources of the slave

2. Offer the resource to the framework, if any, which is furthest below its equal share for its dominant resource. Once offered, the resources in the offer are accounted for in the framework's share.

Chapter 3

Performance of a Single Spark Framework under Mesos

In this chapter we evaluate the Two-level Scheduler consisting of Mesos as Global Scheduler and Spark as framework, which have been described in Chapter 2, in order to answer RQ2: *What is the performance of a single Spark framework under Mesos for interactive queries?* We evaluate the performance through experiments in a real cluster, the DAS-4. We try to answer RQ2 by measuring the performance for different values of multiple configuration parameters. We explore the effects of using Spark in coarse vs fine-grained mode, Spark task parallelism, and allocation interval of Mesos.

We describe the methodology of the experiment in Section 3.1. We describe the data-set used in the experiment in Section 3.2. The experiment setup is described in Section 3.3. In Section 3.4 we present the results of the experiment. Finally, we conclude our findings in Section 3.5.

3.1 Methodology and Metric

Our methodology for testing the performance of Spark-Mesos is as follows. We run our experiments on the DAS-4 cluster. We setup Spark and Mesos and run two representative interactive queries. The queries are run one after another, i.e, we mimic a use-case scenario where there is one end-user. We run the two queries for multiple values of configuration parameters, in order to quantify the performance impact of these parameters. We chose these following configuration parameters as they've shown to influence performance to at least some extent during initial testing:

1. Spark execution mode. Spark can execute in two modes when communicating with Mesos. The two modes differ in how Spark executes a task on a worker node. In fine-grained mode, Spark waits for resource offers for every task. In coarse-grained mode. This difference between Spark modes is further described in Section 2.3.2

2. Allocation interval of Mesos. Mesos begins a new round of offers every second by default. The interval can be decreased such that rounds of offers occur more swiftly after one another.

Table	**Rankings**		**UserVisits**	
Column	name	type	name	type
1	pageURL	VARCHAR(300)	sourceIP	VARCHAR(116)
2	pageRank	INT	destURL	VARCHAR(100)
3	avgDuration	INT	visitDate	DATE
4			adRevenue	FLOAT
5			userAgent	VARCHAR(256)
6			countryCode	CHAR(3)
7			languageCode	CHAR(6)
8			searchWord	VARCHAR(32)
9			duration	INT

Table 3.1: Column names and their respective data-type for the Rankings and UserVisits tables used in the BDB.

3. Task parallelism. By default Spark splits a job into a number of tasks which corresponds to the number of input files. With our input, the default number of tasks for our queries is 200.

We use the *response time* as the metric. The response time is defined as the time between submission of a query to the Spark driver and its completion. The response time gives a good measure of how well an end-user can interactively use the system. If queries have small response times, the user can submit subsequent queries faster after another, leading to higher interactivity between the user and the system.

3.2 Spark Dataset and Queries

The data-set for running the experiments are part of the *Big Data Benchmark (BDB)* [24, 5]. Not all tables and queries which will be explained below will be used in this chapter. However, the complete BDB will be explained as Chapter 4 will assume knowledge of the complete BDB.

The data-set consists of two tables which we can use to perform SparkSQL queries on: A *Rankings* table which contains records consisting of a page URL and pagerank score each, and a *UserVisits* table which contains records which represent structered server logs per URL. The tables are further described in Table 3.1

The BDB includes three queries, each with three variants, which are explained below:

Query 1 filters the Rankings table for specific scores of the pageRank column and outputs the filtered records. There are three specific scores which are used as thresholds for the filtering. The size of the output from this filter query depends on the thresholds used, thus resulting in three variants of query 1, namely, 1A, 1B, and 1C.

Query 2 is an aggregation query over the UserVisits table, which calculates the total advertisement revenue generated for different sourceIPs. There are three variants, 2A, 2B, and 2C

	Input size		Shuffle	Output size		Tables used
	GB	#Records	GB	GB	#Records	
Query 1	6.38	90M				
A			-	2×10^{-3}	33K	
B			-	0.2	3.3M	Rankings
C			-	4.9	89.9M	
Query 2	126.8	775M				
A			6.1	50×10^{-3}	2.1M	
B			12.8	0.76	31.3M	UserVisits
C			14.8	5.8	253.9M	
Query 3	133.2	865M				
A			6.1	1×10^{-3}	1	Rankings, UserVisits
B			10.7	1×10^{-3}	1	
C			38.3	1×10^{-3}	1	

Table 3.2: Summary of the variants of the BDB queries.

for query 2. The difference in these variants is in grouping on different substring lengths of *sourceIPs*. Depending on the length of the substring, the output ranges from two million to 254 million records.

Query 3 joins the two tables, using the *pageURL* and *destURL* columns from the Rankings and UserVisits tables, respectively. The query also aggregates by grouping on the *sourceIP* column, and it shows the total revenue made through advertisements, using the *adRevenue* column. The output is limited to one record consisting of the sourceIP with the highest revenue. The three variants, 3A, 3B, and 3C, differ on the size of the time span between 1980 and three different dates in the visitDate column. The effect of using different time spans, ranging from three months to 30 years, is that the size of data shuffling increases between Spark stages.

An overview of the characteristics of the queries and their variants is given in Table 3.2. We reused an existing implementation [6] of the queries made for Spark and we include them verbatim in Appendix A. Noteworthy are the shuffle sizes of Query 1. The shuffles are non-existent as shuffles are not needed for this type of query. The queries will be used in the experiments in this chapter and Chapter 4 to devise workloads which is further explained in Section 4.2.

3.3 Experiment Setup

We propose a complete setup which will form the basis of experiments in this chapter, as well as for Chapter 4. First, all software described hereafter will be run on server nodes which are

part of the DAS-4 multi-cluster [11]. The DAS-4 is a collaboration effort between universities and is distributed over six geographic locations in the Netherlands. We will use the *TU Delft* and *Vrije Universiteit* sites exclusively, because the hardware available in these clusters is practically the same: Each node has 24 GibiByte (GiB) memory, 8 CPU cores (Dual quad-core) and at least 500 GiB of free hard-disk space. The intra-network communication between nodes is over 20 Gbit InfiniBand, which exposes an TCP/IP interface to applications. The cluster nodes run the CentOS 6 operating system. Second, we use a three tier setup which consists of the following:

1. Mesos as the Global Scheduler (GS)
2. Spark as the general computing framework
3. Network storage, which is exposed by a per-cluster head node and holds the data-set

In total, we use six cluster nodes. One node is dedicated to handle master processes for the Mesos master and Spark's driver process. The remaining five nodes are used as slave nodes by Mesos. A schematic overview of this setup is given in Figure 4.1.

Our Mesos settings are as follows: We use Mesos version 0.21.0, and each node is installed with 24 GiB of memory, though we use 20 GiB as maximum by letting Mesos slaves only report 20 GiB of available memory to the Mesos master. The reason for not allocating the remainder of 4 GiB is that the Operating System (OS) uses memory itself. Mesos slaves are configured to report eight[1] CPU cores (we omit the word "core" hereafter).

Our Spark settings are as follows: We use Spark version 1.3.1. The configuration parameters which are fixed and do not change between experiments, is the amount of 20 GiB memory Spark allocates by running an Spark executor on each Mesos slave. Each executor also allocates one CPU. The setup has five slaves, each having eight CPUs; 40 CPUs in total. However we have 35 CPUs left which can handle spark tasks, since a total of 5 CPUs are used for the executors. Spark uses one CPU per tasks, thus there can be total of 35 tasks running concurrently in the setup. An overview of the setup is given schematically in Figure 3.1. Spark has the ability to cache input data in memory. By enabling caching, relatively slow access to hard-disks is circumvented and results in lower response-times. Before the experiments in this chapter are run, the whole data-set consisting of the *rankings* table is cached from the network storage to memory, by using Spark's caching mechanism. This means that during the experiments, data is not accessed from the network storage.

Our procedure for evaluating is running queries 1A and 1B of the BDB and measure the response-times. We run the queries 20× for each configuration parameter. We use these two queries, because they finish within seconds, therefore they can be considered interactive queries. This is contrast to query 1C, query 2, and query 3, which run in the order of minutes. The output of the queries is sent back to the Spark driver, which mimics the use-case scenario when an end-user would use the system interactively, since that user would want to see the results on his local system.

[1] Mesos defaults to using the number of logical cores, i.e., it would report 16 available cores per node, as a physical core counts as two cores because of Hyperthreading

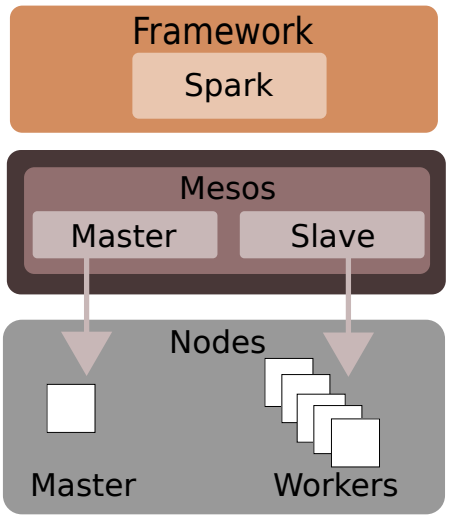


Figure 3.1: An overview of the experiment setup, showing the software and hardware components.

Parameter	Value	Factor
Spark caching	Enabled	X
Spark mode	Fine-grained / Coarse-grained	✓
Spark parallelism (#partitions)	50 to 200	✓
Mesos allocation interval (ms)	12 to 1000	✓

Table 3.3: Parameters and their values used in the experiment

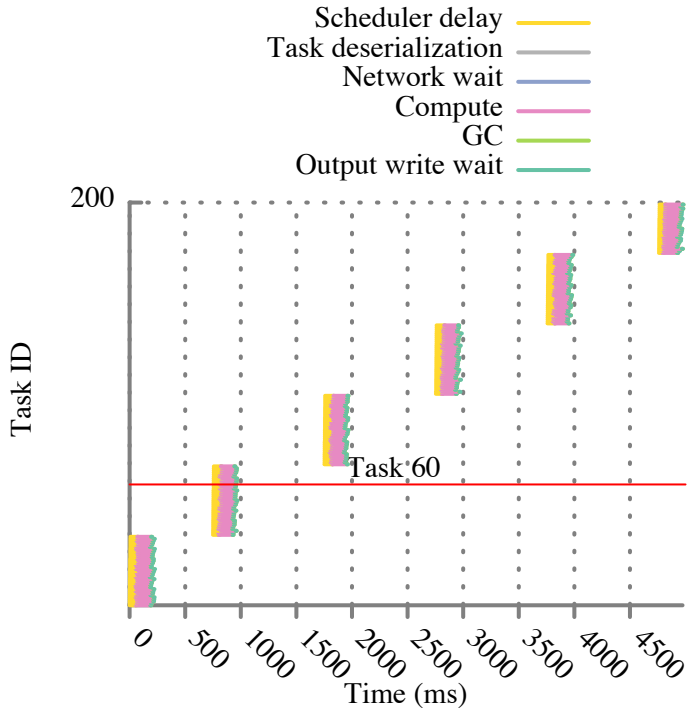
3.4 Experiment Results

We have run experiments for the configuration parameters and their values given in Table 3.3, which will be addressed in their respective subsections hereafter.

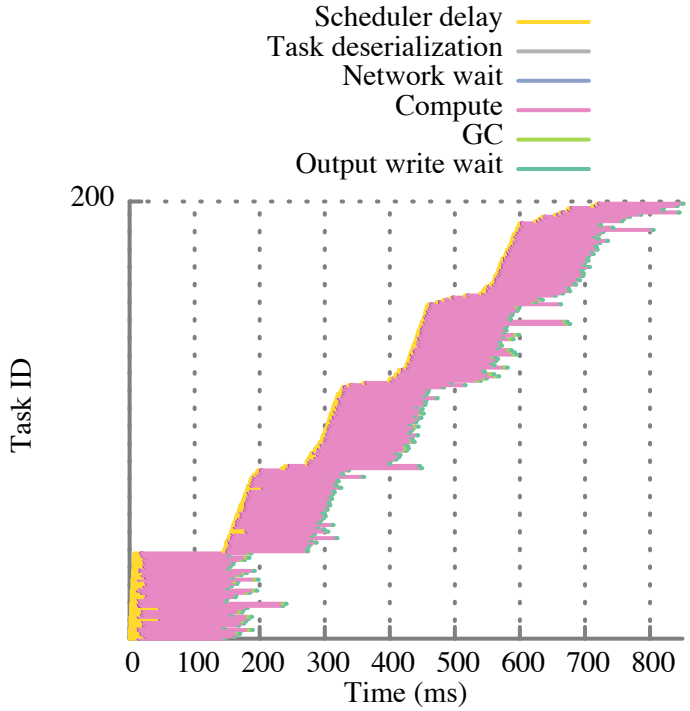
3.4.1 Fine-grained vs Coarse-grained mode

First, we show the time decomposition of two representative, out of 20 executions of queries 1A in Figure 3.2. The sub-figures depict the same experiment, but with different parameter settings for the Spark mode, i.e., fine-grained and coarse-grained.

The horizontal axis shows wall-time and the vertical axis shows tasks. Thus, the complete job exists of 200 tasks and the runtimes of individual tasks are shown as horizontal bars. For example, task with ID 60 (highlighted in the figure), starts at roughly 750 ms and completes before 1 second. The runtime is further composed of scheduler delay, task deserialization, network

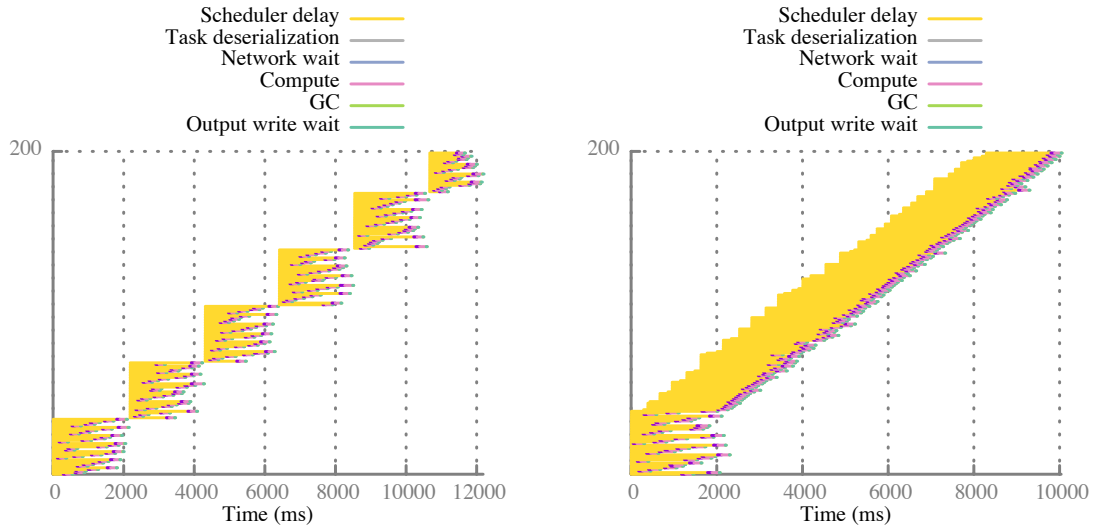


(a) The time decomposition of query 1A under Spark-Mesos in fine-grained mode



(b) The time decomposition of query 1A under Spark-Mesos in coarse-grained mode

Figure 3.2: The time decomposition of all tasks of a Spark job running query 1A of the BDB under Mesos in fine-grained (a) and coarse-grained (b) mode. (Note the different time scales on the horizontal axes.) The horizontal bars represent tasks are composed of more detailed timing information as indicated by the legend. As an example, the marked task 60 in (a), starts at 750 ms and finishes at 1 s.



(a) The time decomposition of query 1B under Spark-Mesos in fine-grained mode

(b) The time decomposition of query 1B under Spark-Mesos in coarse-grained mode

Figure 3.3: The time decomposition of all tasks of a Spark Job running query 1B of the BDB under Mesos in fine-grained (a) and coarse-grained mode (b) mode. (Note the different time scales on the horizontal axes.)

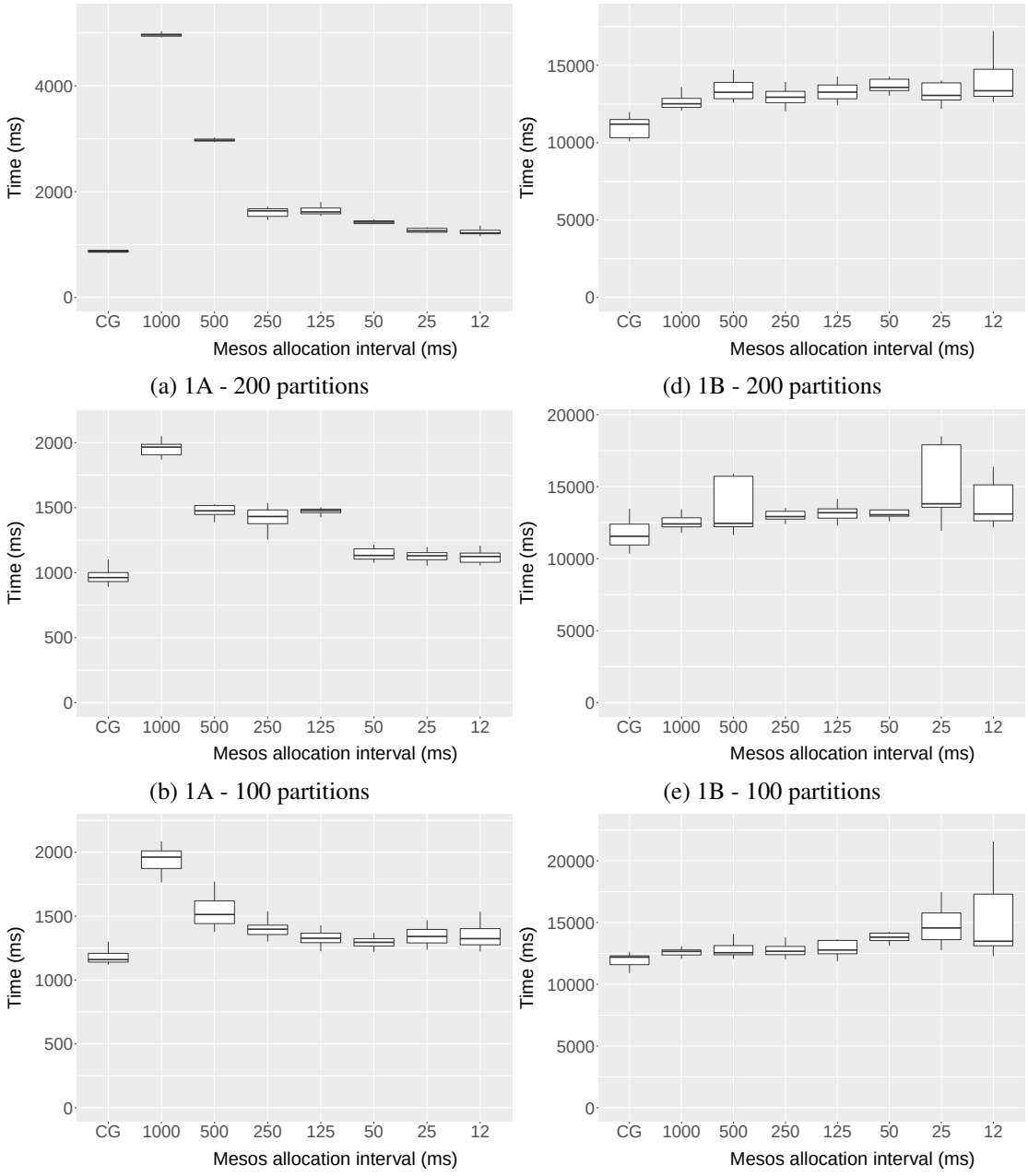
wait, compute, garbage collection, and output write time. Task 60 is in the second *wave* (see Section 2.3.2) together with 34 other tasks.

There are things to note. First, the difference between the total job response times which are around five seconds for fine-grained mode, and less than one second for coarse-grained mode. Second, in fine-grained mode, the utilization is lower, as indicated by the time intervals in which no tasks are running. The waves start at one second intervals, which corresponds to the default setting of the allocation interval of Mesos, which has one second period. The allocation interval determines the frequency at which Mesos master offers resources to frameworks. The speedup of coarse-grained compared to fine-grained is $4965\,\mathrm{ms}/870\,\mathrm{ms} \approx 5.7\times$

In the previous experiment we used query 1A of the BDB to measure response times. We do the same experiment with query 1B of the BDB. Query 1B has a larger result output (see Table 3.2), which needs to be sent to the Spark driver. In Figure 3.3 we can see that scheduler delay, is using a larger fraction of the total runtime of a job. In Spark, scheduler delay includes the time needed for sending task data (input or output) from the driver to the executors and vice versa. We see that with default settings the speedup between fine-grained and coarse-grained is smaller than with query 1A; the speedup for coarse-grained vs fine-grained is $12\,\mathrm{s}/10\,\mathrm{s} = 1.2\times$.

3.4.2 Allocation interval of Mesos

To determine the influence of the allocation interval, we run another experiment shown in Figure 3.4a. The figure shows eight box-and-whisker plots, each representing 20 iterations of query



(a) 1A - 200 partitions

(d) 1B - 200 partitions

(b) 1A - 100 partitions

(e) 1B - 100 partitions

(c) 1A - 50 partitions

(f) 1B - 50 partitions

Figure 3.4: The job response times for query 1A, (a), (b), (c), and 1B, (d), (e), (f), under Spark-Mesos for three different partition sizes. Each figure compares coarse-grained mode (leftmost boxplot labeled *CG*) to fine-grained mode, for different values of the Mesos allocation interval parameter. Whiskers of the box-plots show the 5th and 95th percentile.

1A for different settings. The leftmost boxplot shows the response times using coarse-grained (CG) mode. The seven rightmost boxplots show the response time in fine-grained mode with decreasing allocation interval size. The response time with coarse-grained and fine-grained mode is mostly under one and five seconds respectively (which corresponds with Figure 3.2). We see that changing the default one second allocation interval to $500\,\mathrm{ms}$ and $250\,\mathrm{ms}$, decreases response time. Decreasing the allocation interval even more shows diminishing returns. By using a lower allocation interval, we decrease the median response time from $4.9\,\mathrm{s}$ to $1.2\,\mathrm{s}$.

In Figure 3.4d we depict the boxplots for the response time over 20 iterations for query 1B, and varying values of the Mesos allocation interval. In contrast to the improvement shown for query 1A, the response times are slightly larger when using a smaller allocation interval than the default 1 second.

3.4.3 Task Parallelization

Another configuration parameter besides the allocation interval of Mesos, is the number of partitions used in Spark. The partition size influences the amount of waves needed to complete one query. With default settings we needed $\lceil 200/35 \rceil = 6$ waves, see Figure 3.2a. In Figures 3.4b and 3.4c, and Figures 3.4e and 3.4f, we change the number of partitions Spark creates for the data set of the BDB, for query 1A and 1B respectively. This does not influence the total processed amount of data, and effectively reduces the number of tasks, but increases the work done per task.

For query 1A we see that reducing the number of partitions to 100, from the default 200, improves response time overall. The reason for this improvement is due to fewer waves needed. Coarse-grained mode has been the fastest option so far, and it remains so compared to fine-grained mode. However, absolute performance degrades when choosing 50 partitions. In this case the limited parallelism of 50 tasks is resulting in lower utilization of the system during its second wave of only 15 tasks $(50 - 35)$. Apparently this under utilization degrades performance more than the extra overhead of running more tasks. In the case of fine-grained mode, the median response time decrease from 4.9 seconds, when using 200 partitions, to 1.9 seconds for 100 partitions. Using 50 partitions for query 1A does not yield more performance gain compared to 100 partitions.

For query 1B we do not see any significant performance performance gain when using less partitions, in fact the response times increase slightly. However, for the combination of lower allocation intervals, $25\,\mathrm{ms}$ and $12\,\mathrm{ms}$, and less partitions, the boxplots show a larger range for the IQRs.

3.5 Conclusion

In this chapter we have run a configuration parameter study for the Spark framework running under the Mesos global scheduler. The parameters we have studied provide practical insights on how Spark-Mesos configuration parameters can influence the runtimes of interactive queries on Spark-Mesos.

We run two types of queries on a synthetic dataset, which has also been used in previous work, to evaluate the runtime for various values of configuration parameters. We found that the difference in performance of Spark-Mesos using fine-grained mode, compared to coarse-grained mode, can be more than $5\times$ when using our queries with default settings for Mesos and Spark. The main issue for achieving performance for Spark in fine-grained mode is that it relies on the individual task runtime.

We found two ways to increase the performance when using fine-grained mode. First, we set smaller allocation intervals for Mesos. This results in higher utilization, because the allocation interval parameter influences Mesos to make offers more often enabling higher throughput tasks, which increases utilization in the case of small tasks. Second, we repartitioned the input to less partitions than the default, resulting in larger partitions. As Spark executes one task per partition, less tasks needed to be executed, whereas the utilization increased since runtimes of tasks take longer as the input data was now larger as well.

Overall, Spark's coarse-grained mode remains the most efficient solution. By performing this parameter study for Spark-Mesos we have answered the second research question RQ2.

Chapter 4

Performance Balance among Multiple Spark Frameworks under Mesos

In this chapter we evaluate the ability of the proposed Two-level Scheduler (TLS), consisting of Spark on Mesos, to be able to maintain performance balance to answer RQ3: *Can we achieve a performance balance for non-interactive data-analytics queries between multiple Spark frameworks running under Mesos?* Performance balance between frameworks on the same cluster, or datacenter, can be desirable in the case of workloads that vary over time. Giving static partitions of the cluster resources to frameworks which run the workloads may be sub-optimal. Instead of using static partitions, we will use the mechanisms of Mesos to allocate resources to frameworks dynamically.

We first describe the methodology for setting up an experiment for achieving performance balance in Section 4.1. We then describe our method of generating workloads in Section 4.2, which are used in the experiments. We describe and show results for static allocation experiments and dynamic allocation experiments in Sections 4.3 and 4.4, respectively. The results of the experiments are summarized in Section 4.5

4.1 Methodology and Metric

Our general methodology for achieving performance balance between multiple frameworks is as follows. We run our experiments in a setup which is similar to the setup used in Section 3.3. However, we will now use multiple Spark frameworks. We generate multiple workloads with arrival patterns such that the workloads have time-varying loads.

For the experiments in this chapter, the workloads consist of data-analytics jobs, which are larger in terms of job response time compared to the interactive queries in Chapter 3. Furthermore, the output of these jobs does not get sent back to the Spark driver, but gets saved to the Hadoop Distributed File System (HDFS) layer.

We use *job slowdown* as metric, which is defined as the response time for a job in a busy system, over the response time in an empty reference system. Response times in a busy system can be larger than in the reference system as waiting time in the framework's job queue can occur.

The metric for performance balance is defined as the ratio among the median job slowdowns per workload. During the workload generation we use *service time* as a metric for query size. The service time is the total amount of time needed to execute a query by taking the sum of the execution times of its individual tasks.

4.2 Workload Generation on Reference Setup

In this section we devise a reference system on which we will run queries of the Big Data Benchmark (BDB) [5]. The response times for the queries will be used as reference times in order to generate workloads based on those response times. We describe the reference system in Section 4.2.1 and show the results in Section 4.2.2.

4.2.1 Experiment Setup

We use the experiment setup from Chapter 3 as a basis for the reference setup hereafter. The setup will be run on server nodes which are part of the DAS-4 multi-cluster [11]. However, in this chapter we do not run interactive queries. Since we assume non-interactive querying, we do not cache the dataset, instead we read input from HDFS for every query and write the result back to HDFS. We use a three tier setup which consists of the following:

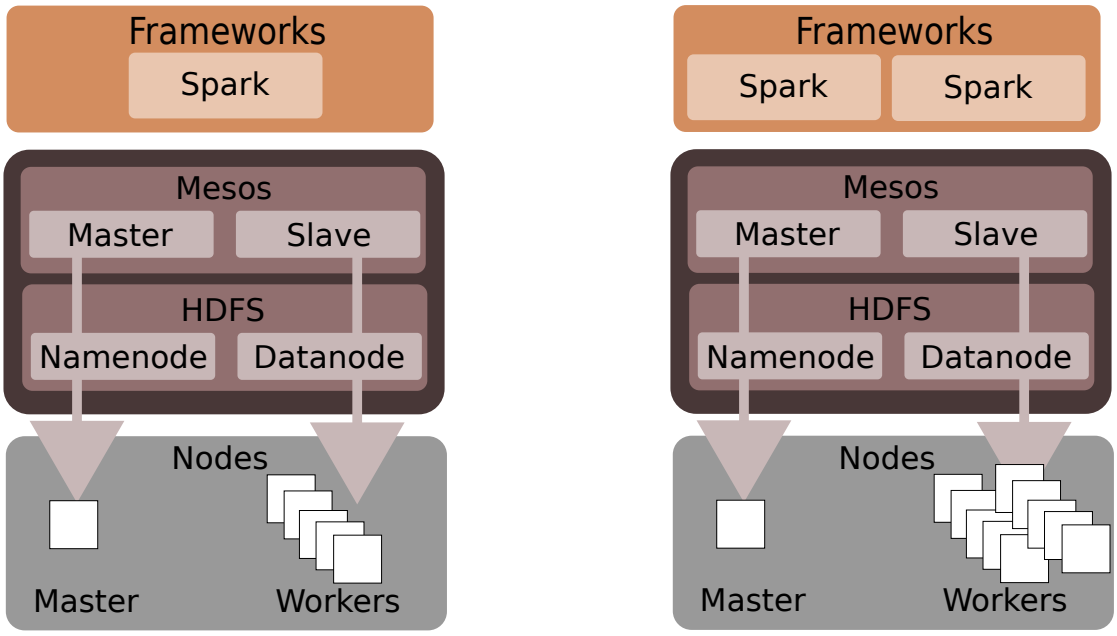
1. Mesos as the Global Scheduler (GS)
2. Spark as the general computing framework
3. HDFS as the data layer.

In total we use six cluster nodes. One node is dedicated to handle master processes for the Mesos master, Spark's driver process, and the HDFS namenode. On each of the remaining nodes, we run mesos slaves, thus indirectly Spark executors, and HDFS datanodes. A schematic overview is of this setup is given in Figure 4.1.

Our Mesos settings are as follows. We use Mesos version 0.28.0, and each node is installed with 24 GibiByte (GiB) of memory, though we use 20 GiB as maximum by letting Mesos slaves only report 20 GiB of available memory to the Mesos master. The reason for not allocating the remainder of 4 GiB is that the Operating System (OS) uses memory itself. Mesos slaves are configured to report eight[1] CPU cores (in the remainder of this thesis we simply refer to CPUs). Our Spark settings are as follows. We use Spark version 1.5.1. Spark is run is configured to run in fine-grained mode and the Spark executors use 5 GB. HDFS is setup with 128 MiB blocks. Before an experiment runs, we assume the complete dataset of the BDB is present on HDFS.

Our procedure for evaluating reference times for the BDB queries is to measure the mean response time of five executions for each query. We flush the operating system file cache before each execution. The characteristics of the queries have been described in Table 3.2.

[1] Mesos defaults to using the number of logical cores, i.e., it would report 16 available cores per node, as a physical core counts as two cores because of Hyperthreading



(a) Single-framework setup

(b) Multi-framework setup

Figure 4.1: An overview of the experiment setups, showing the software and hardware components for the single-framework (a) and multi-framework setup (b).

4.2.2 Reference Experiment Results and Workload Distribution

Table 4.1 shows the service times and response times for the queries of the BDB. Based on the service times of the queries we distinguish three classes of query sizes: *small*, *medium*, and *large*. The small class contains queries that have service times lower than 1000 s, the medium class contains queries that have service times lower than 7000 s, and the large class contains query 3C which has a service time larger than 7000 s.

Previous work [25, 12] shows that small queries represent the majority of job submissions in typical data-analytics workloads. Although our specific data-size does not correspond to the queries used in the previous work, we will use a job-size distribution of 75%, 20%, and 5% for small, medium, and large jobs, respectively. This distribution corresponds well to the distribution of job durations in [25]. The queries are uniformly chosen within their size-classes. With this distribution, the average query time ($\overline{T_Q}$) is calculated as follows:

$$\begin{aligned} \overline{T_Q} &= 0.75 \cdot T_{small} && + 0.20 \cdot T_{medium} && + 0.05 \cdot T_{large} \\ &= 0.75 \cdot 479.0s && + 0.20 \cdot 6496.4s && + 0.05 \cdot 13520.3s \\ &= 2334.5s. \end{aligned} \tag{4.1}$$

In the remainder of this chapter we will use the above average query time to generate workloads with the required utilizations.

	Service Time (s)	Response Time (s)	Size	Avg service time (s)
Query 1				
A	450.0	14.0	Small	479.0
B	435.4	13.8		
C	551.7	20.3		
Query 2				
A	5815.6	176.2	Medium	6496.4
B	6420.2	190.5		
C	7170.5	212.0		
Query 3				
A	6285.8	191.4		
B	6789.8	203.1		
C	13520.3	377.2	Large	13520.3

Table 4.1: The service times and response times for the queries of the BDB. The queries are classified in three classes: small, medium, and large, and classes small and medium have multiple types of queries.

4.3 Static Allocation

In this section we run and discuss experiments that measure the effect of sharing resources over multiple frameworks when each runs a workload. During the experiments we assume that we know the imposed loads of the workloads. We run the following experiments:

1. Equal load experiment. Using a Poisson process, we emulate user submitted queries to *one* Spark framework using five worker nodes, called the single-framework experiment. We compare the results from the single-framework experiment to the multi-framework experiment in which two frameworks receive user submitted queries. The number of worker nodes is doubled. However, the imposed load for the total system remains the same as each framework runs the same workload as used in the single-framework experiment. By comparing the single-framework to the multiple-framework system, we aim to show the benefits of sharing resources between multiple frameworks for time varying workloads.

2. Unequal load experiment. In this experiment we reuse the same multi-framework setup. However, the two frameworks in this setup will run workloads with unequal imposed load. We run this experiment to gain insight on how Mesos distributes resources to the Spark frameworks when they impose different loads.

3. Unequal load aware experiment. Similar to the previous, however, we instruct Mesos to allocate resources in a non-default manner. Mesos will be configured to offer resources to

frameworks corresponding to their imposed loads on the system, e.g., if framework A has an imposed load of 0.2 on the system, and framework B has an imposed load of 0.4, then Mesos will offer twice the amount of resources to framework B, compared to A.

These experiments, in particular the last, function as a base-line for the dynamic experiments in the next section.

4.3.1 Equal Load Experiment

The purpose of this experiment is to measure the relative performance of two types of systems. The first system consists of two single Spark frameworks which each can assign their jobs to their own set of 5 nodes. The other system consists of two frameworks as well, but they share 10 nodes to which they can assign their job tasks. We will refer to these two systems as the *single-framework* and *multi-framework*, respectively. In the experiment setup, we describe three workloads with different imposed loads. We run each workload on both systems, and we first report the measured load to check if our imposed load are actually resulting in the intended load. Second, we will compare the job slowdowns achieved for the single-framework and multi-framework setup.

Experiment Setup

For the single-framework setup, we reuse the reference system of Section 4.2. For the multi-framework setup, we extend the reference system by increasing the number of Spark frameworks to two and the number of worker nodes to 10. We also add one extra node, on which the second Spark framework will run the driver process, so that each Spark driver will be on its dedicated node. In total we use 12 nodes for this setup.

In the multi-framework setup the two frameworks both can get offers from Mesos for any of the 10 worker nodes, i.e., the frameworks are not restricted to a subset of nodes for placing their tasks.

In order to impose a workload on the system, we emulate job arrivals by using a Poisson process. The jobs are queries, chosen randomly from the BDB according to a distribution depending on the job size. Using the measured service times we generate three workloads, W_l, W_m, and W_l , each spanning a six hour period, with imposed loads of 0.5, 0.7 and 0.9, respectively on 5 nodes. The expected number of jobs and arrival rates as a function of $\overline{T_Q}$ for the three workloads are described in Table 4.2. The arrival rate is the number of job submissions per second. The arrival rates for the different workloads are calculated from the general formula for the utilization ρ:

$$\rho = \frac{\lambda}{(c \cdot \mu)}, \tag{4.2}$$

where μ is the service rate, i.e., the average number of jobs that can be processed by a single CPU, which is the reciprocal of $\overline{T_Q}$, and c is the number of CPUs in the system.

In the experiments in this chapter, if two frameworks are running the same workload, e.g., W_h, they each refer to an independent copy with their own independent arrival process.

Workloads Characteristics			
Workload	Imposed Load	Arrival Rate (s^{-1})	#Jobs
W_l	0.5	0.008 57	185
W_m	0.7	0.011 99	259
W_h	0.9	0.015 42	333

Table 4.2: The characteristics of three generated workloads that each span six hours.

Results and Discussion

We first show the actual measured load for each workload in order to verify that the loads represent the load which was imposed. In Table 4.3 we see that the measured loads are approximately 90% of the imposed load for W_l and W_h. However, for W_h the measured load is 90% of the intended imposed load for the single frameworks, but for the multi frameworks the measured load is larger than the imposed load. After examining the metrics, we conclude the difference between imposed and measured is due to OS-cache which holds parts of data originating from the HDFS in memory such that subsequent access to data is faster. However this does not seem to hold for W_h. Furthermore, workload W_h has shown to be a too high imposed load as the queue size in Spark grew unbounded. After the six hour workload, the system needed more than an half our to finish the workload. We will show the results for W_h but they are not representative. In a real environment where these high loads are encountered, a reasonable solution would be to acquire more resources, i.e., to add more servers to the cluster.

To remind the reader, in the reference experiment of Section 4.2, we explicitly flushed caches in order to get reproducible results. Flushing caches on OS level in these workloads with arrivals is not possible as the flushing itself takes time, which could influence the start times of other jobs in the queue. Except for the W_h workload, the differences between the two systems in terms of measured load do not deviate greatly from each other, and the comparison between the two should be representative.

In Table 4.4 we now show the average and median job slowdowns for complete workloads for both the single and multi framework setup. Omitting W_h from the discussion, overall we can see that the mean and median job slowdowns are better for the multi-framework setup. The difference in slowdowns is more pronounced for the W_l workload then for W_m, which is expected, because in the case of W_l there are more opportunities for either framework to acquire more resources than their guaranteed half of the resources if the other framework is not using their resources. This technique is called statistical multiplexing in general and even allows for median job slowdowns smaller than one. This implies that the workload has half of its jobs run faster than on the reference system. This is possible because in the multi-framework setup a job can in theory have twice the number of tasks running concurrently.

In Figure 4.2, we show the job slowdowns for every type of query and aggregated for all queries (leftmost boxplot). The job slowdowns are indeed smaller for the multi-frameworks compared to the single-frameworks, for W_l and less pronounced for W_m workload.

When comparing queries we can see a trend of relatively high job slowdowns for all queries

	Workload		
	W_l	W_m	W_h
Single-framework			
F_1	0.46	0.60	0.81
F_2	0.46	0.63	0.84
Multi-framework			
F_1	0.50	0.63	0.91
F_2	0.48	0.63	0.96

Table 4.3: Measured loads for the workloads, for each Spark framework and for the single and multi-framework system.

Job Slowdown Characteristics for Equal Workloads						
	Mean			*Median*		
	W_l	W_m	W_h	W_l	W_m	W_h
Single-framework						
F_1	7.3	10.3	31.0	1.12	3.58	14.31
F_2	7.0	10.6	59.1	1.21	3.73	40.25
Multi-framework						
F_1	3.8	9.2	59.8	0.80	2.69	19.01
F_2	4.3	8.1	85.0	0.90	2.04	36.92

Table 4.4: The mean and median slowdowns of all queries for each workload.

Unequal Load Workload Characteristics			
Workload	Imposed Load	Arrival Rate (s^{-1})	#Jobs
1:2 ratio			
$W_{2,l}$	0.234	0.008 002	173
$W_{2,h}$	0.467	0.015 99	345
1:4 ratio			
$W_{4,l}$	0.14	0.004 797	104
$W_{4,h}$	0.56	0.019 19	415

Table 4.5: Workload characteristics for two instances of the unequal load experiment with a total imposed load on a system of 10 nodes of 0.7

of type Query 1 compared to the other queries. This trend shows for every workload. The reason for these high job slowdowns is that query 1 is in the order of seconds. When a larger query is in front of query 1 in the FIFO of Spark, query 1 will have a high waiting time before it will be processed, thus resulting in a high job slowdown.

4.3.2 Unequal Load Experiment

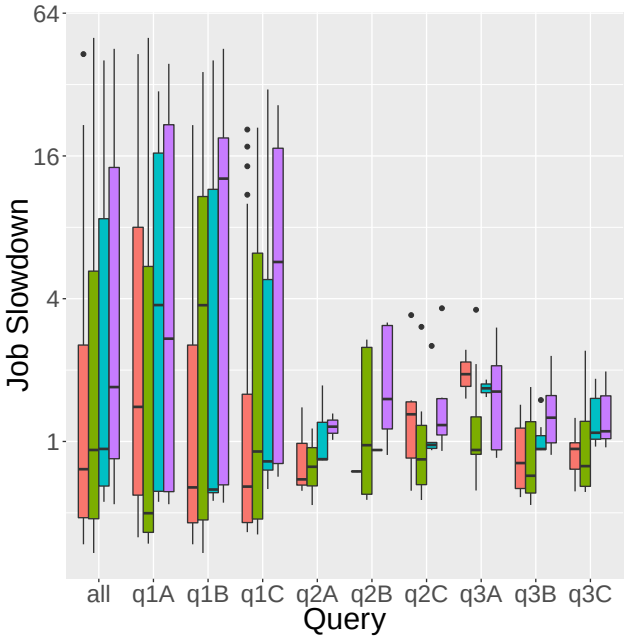
In the previous experiment we measured the job slowdowns for two frameworks with equal imposed loads. In the following experiment we use the same multi-framework setup as the previous experiment, but with different imposed loads. The sum of the loads imposed by two frameworks is 0.7, for a system of 10 nodes, but the imposed load per framework differs by a certain ratio. We use two ratios 1:2 and 1:4 to form two instances of this experiment.

The generation of these workloads is done similarly as the workloads so far, described in Section 4.2. We generate two pairs of workloads for two experiment instances. The workload characteristics are described in Table 4.5.

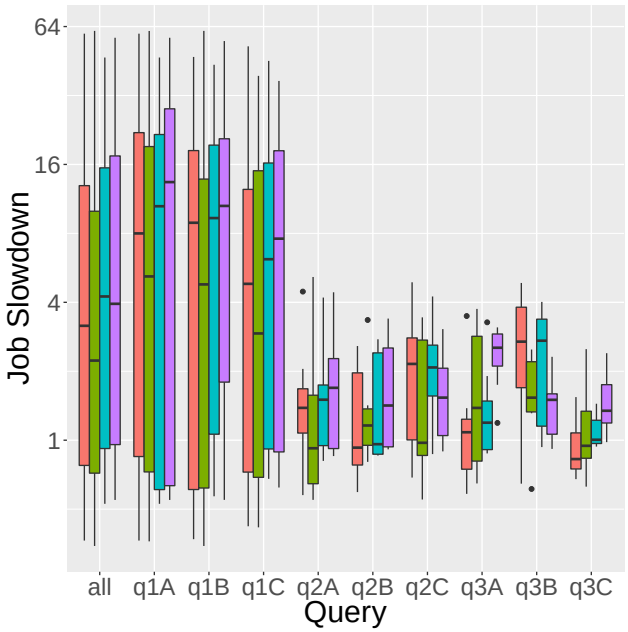
The results of the two experiment instances are given in Table 4.6. We can see that there is a difference between the job slowdowns of the frameworks for both ratios. The difference in job slowdown for the 1:4 ratio experiment is more pronounced than for the 1:2 ratio.

In Figure 4.3 we show the slowdowns for each query of the experiment for the 1:4 ratio. We can see that difference between frameworks in terms of job slowdown holds for all type queries. The reason for this difference in job slowdowns between frameworks is that Mesos offers resources equally by giving frameworks roughly 50% of the cluster resources at any given time that a framework accepts it, i.e., when it has at least one job to run. In this experiment this means that Mesos gives resources to the framework with smaller imposed load at the expense of the framework with larger imposed load.

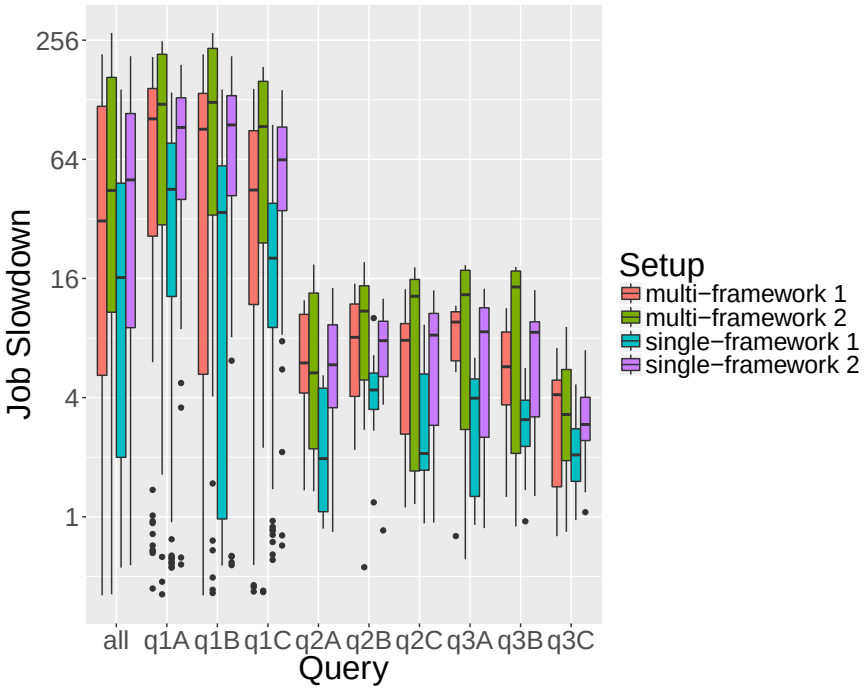
We conclude this experiment by noting that without instructing Mesos to offer resources in a specific manner, its resource allocation, Dominant Resource Fairness (DRF), is favoring frameworks which have a smaller imposed load. In the next section we instruct Mesos to offer resources in the same ratio as the imposed load.



(a) Imposed load of 0.5, running workload W_l



(b) Imposed load of 0.7, running workload W_m



(c) Imposed load of 0.9, running workload W_h

Figure 4.2: Boxplots of the per-query type job slowdowns per query type, of the single-framework and multi-framework experiments for different imposed loads. The vertical axes are in logarithmic scale.

Job Slowdown Statistics				
Framework	Workload	Mean	Median	95th percentile
F_1	$W_{2,l}$	5.28	1.20	21.97
F_2	$W_{2,h}$	13.52	4.35	52.19
F_1	$W_{4,l}$	3.11	1.00	10.63
F_2	$W_{4,h}$	47.86	29.19	204.01

Table 4.6: Statistics of the job slowdowns for two ratios between the imposed loads of two frameworks in the unequal load experiment.

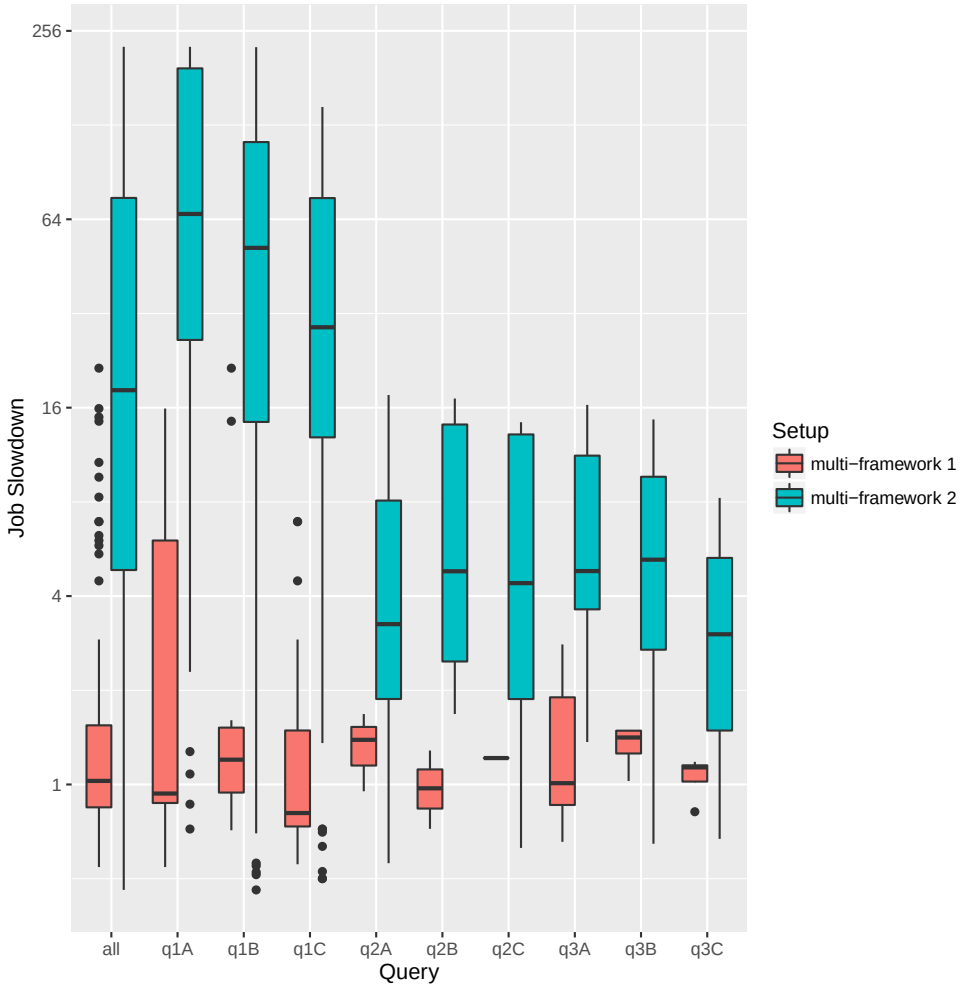


Figure 4.3: The job slowdowns for the unequal load experiment, where the loads differ in a 1:4 ratio. The combined load of the frameworks is 0.7 on a 10 node system.

Job Slowdown Statistics for Unequal Aware Workloads				
Framework	Workload	Mean	Median	95th percentile
F_1	$W_{2,l}$	10.61	2.54	36.82
F_2	$W_{2,h}$	6.22	1.76	23.49
F_1	$W_{4,l}$	68.07	13.94	292.67
F_2	$W_{4,h}$	9.03	2.55	39.19

Table 4.7: Job slowdown statistics for two instances of the unequal load aware experiment. Each instance has a different ratio between imposed loads by two frameworks. During the experiment Mesos allocated resources to frameworks in the same ratio of the loads imposed by the frameworks.

4.3.3 Unequal Load Aware Experiment

In the previous experiment we introduced unequal workloads, but we did not change any configuration parameters. We aim to have a better performance balance by taking advantage of our a priori knowledge of the imposed loads of the workloads. For this experiment this requires us to allocate resources to the frameworks in the same ratio as their imposed loads. We can do so by setting the weights for frameworks in DRF, in the same ratio of the imposed loads (see Section 2.4). By setting weights for DRF, the allowed resource shares for frameworks can be influenced to differ from the default fair share.

In total we have four attempts in which we try to achieve a situation where Mesos offers resources in the intended ratio between frameworks, corresponding to their imposed load. Our first attempt is setting the weight parameters in DRF, but will show to be inconclusive. We describe the other attempts in their respective paragraph hereafter.

Attempt one (A1) Rerunning the previous experiments with the corresponding allocation weights ratio in DRF, we get the results as shown in Table 4.7. For the 1:2 workload the difference between the median and average job slowdown of both frameworks are closer to each other than the corresponding slowdowns for the 1:2 ratio in Table 4.6. However, this improvement does not show in the 1:4 workload experiment. In fact, in contrast to the previous experiment of Section 4.3.2, the smaller framework (F_1), which ran $W_{4,l}$, has a significantly larger job slowdown than the larger framework running $W_{4,h}$. We can conclude that F_1 does not get enough resources to cope with its workload.

We expected both instances of the experiment to show a better performance balance, i.e., having a smaller difference in median and average job slowdown between the two frameworks. Further investigation shows that in the 1:4 workload experiment, framework F_1 with smaller imposed load ($W_{4,l}$) does not get CPU resources by Mesos as long as F_2 has waiting jobs. This under-allocation stems from the DRF policy which takes into account the *dominant* resource of a framework. When Spark starts, it places executors on every node it gets offers to. An executor is configured to take up 5GiB. Thus, per node an executor uses 25% of the memory

resources, and this holds for every node in the cluster, thus both frameworks each use 25% of the memory resource in the cluster. This memory is not relinquished by the frameworks until the Spark application ends, which is after running the complete workload. This results in DRF considering that F_1 is always achieving its fair share, since the 1:4 ratio of the weights in DRF dictates that F_2 is allowed to have a dominant resource share of $4\times$ 25%, i.e., starvation for F_1 occurs, because it will not get CPU resources, unless F_2 has an empty job queue.

Our following attempts will try to address the problem of the 1:4 workload. However, we are content with the attained performance balance for the 1:2 workload, which is within approximately $1.44\times$ for the median job slowdown and $1.70\times$ for the mean job slowdown.

Attempt two (A2) In order to test the hypothesis that the dominant resource of memory for F_1 is resulting in under-allocation of CPU resources in the case of the 1:4 workload experiment, we propose to inflate the amount of memory in the cluster by configuring Mesos to report $10\times$ more memory in the system. Consequently, DRF will consider F_1 to have only a 2.5% share of total cluster memory. This means that once F_2 rises above a (dominant) CPU share of 10%, F_1 will begin receiving offers as well. The previous experiment is rerun and the results are shown in Table 4.8

We see that inflating the amount of memory in the system results in a better performance balance, as the mean and median job slowdowns are much closer to each other than in our first attempt. However, by inflating the amount of memory, we only cater for our specific use-case of two frameworks which use a fixed amount of memory. In a real-life setting, the wrong amount of memory can be problematic for frameworks which have a variable usage of memory, therefore this is not a satisfactory solution.

Attempt three (A3) Another option besides inflating the memory is to compensate the allocation ratio such that F_2 can not have a larger cluster share than 80% CPUs, which is what we want to achieve in the first place with a 1:4 ratio. In order to do so, we would need to compensate the allocation weights ratio in DRF. As explained above, Spark uses a fixed amount of $5\,\mathrm{GiB}$ per worker node, resulting in a share of 25% of the cluster's memory. This means that F_1 is always above its guaranteed share of 20% and is unlikely to receive CPU offers. Because the weight ratio in DRF is set to 1:4 for F_1 and F_2 respectively, and F_1 is using 25%, F_2 will be offered a theoretical $4\times$ the amount of resources of F_1, i.e., F_2 will be offered *all* resources.

In order to make the dominant share of F_2 at most 80%, thus leaving 20% for F_1, we need to alter the allocation weight ratio to $80\%/25\% = 3.2$. We *expect* that with a ratio of 1:3.2, DRF will offer 80% of the cluster's resources at maximum to F_2. The remaining 20% of resources should then be offered to F_1.

The results of the experiment with the compensated allocation ratio are in Table 4.8. Compensating the ratio does not have the desired effect and investigation shows that Mesos under-allocates the smaller framework. We found that the DRF implementation accounts memory immediately to a framework's total resources, which nullifies our attempt of using a compensated allocation weight ratio in the following way:

1. Before Mesos starts a new offer round, framework F_1 and F_2 each have a dominant share of 25% of the clusters memory.

Job Slowdown characteristics				
Configuration	FW	Median	Mean	95th percentile
A2: Inflate memory	F_1	6.49	22.00	95.31
	F_2	7.76	27.50	112.89
A3: Compensate ratio	F_1	13.92	68.80	315.53
	F_2	6.27	23.27	107.72
A4: Compensate ratio + spread allocation	F_1	6.77	25.88	121.95
	F_2	8.20	26.77	102.98

Table 4.8: Job slowdown statistics for three different attempts of the unequal load aware experiment. Each experiment has a total imposed load of 0.7, and two frameworks impose this load in a 1:4 ratio.

2. The offer round starts and for each mesos slave, DRF accounts the available resources of that slave to the framework furthest below its weighted fair share. The available resources, disregarding CPU resources, are $10\,\mathrm{GiB}$ per mesos slave, i.e., 5% of the total memory in the cluster.

3. At one point in time during the offer round, if F_1 is furthest below its weighted fair share, F_1 will be accounted the $10\,\mathrm{GiB}$ of a mesos slave, i.e., 5% of the cluster's memory. At this point, F_1 will now have a dominant resource share of 30%.

4. If there are more mesos slaves with available resources, F_1 will never be furthest below its weighted fair since, since F_2 is now allowed to grow to $3.2\times$ the dominant share of 30% of F_1, thus F_2 will be offered 96% of the cluster resources, instead of the 80% we intended.

5. If there are no more slaves with available resources, the resource offers are sent to the frameworks.

Similar to attempt A1, we suffer from under-allocation for the smaller workload run by F_1, because F_1 will only receive resources from one slave at maximum, per offer round.

Attempt four (A4) Our last attempt to achieve performance balance is to circumvent the problem encountered in attempt A3 that F_1 is unlikely to receive more than one offer per allocation round. The underlying problem for our use-case is that DRF accounts memory resources immediately to frameworks during an offer round, and only at the end of the offer round sends the offers. Thus, even if frameworks will not use the memory they are offered, it can increase their dominant resource share for the duration of the offer round.

Our proposed solution is to alter the implementation of DRF to send offers immediately after accounting resources, and we instruct DRF sleep for a relatively small time before continuing with other mesos slaves in the offer round. During the sleep command of DRF, frameworks

will have the opportunity to reject the resources which they do not need. In our case this means that after F_1 is accounted $5\,\mathrm{GiB}$ of extra memory, raising its dominant share to 30%, it will be offered that memory, and it can reject it immediately lowering its dominant memory back to 25%, though it does accept any CPU resources in the offer. After the sleep period, DRF will continue its offer round, and F_1 will still be viable for resources offers.

The results of running the experiment, with the above proposed change to the implementation of DRF, is in Table 4.8. We can see that our changes to the Mesos allocator have had the desired effect of bringing job slowdown performance balance between the two frameworks. The performance is only slightly worse than performance balance measured using the memory inflation technique. We conclude that this approach has the desirable properties of giving good performance balance without resorting to the memory inflation of attempt A2.

4.4 Dynamic Allocation

In this section we explore the ability to respond to changes in workload in order to achieve a performance balance between frameworks. In the previous section we have shown that a performance balance can be achieved, which is within a factor of 2 for the job slowdown statistics, by setting the allocation weight ratio of DRF, equal to the ratio of the imposed loads between frameworks. However, it is not always known in advance what the imposed load of a workload is going to be. We propose to use a feedback mechanism to react to the online performance of the frameworks.

We will discuss the feedback mechanism and accompanying policy in Section 4.4.1. In the two subsection thereafter we will describe two experiments and their results in the case of using two and three frameworks competing for resources.

4.4.1 Feedback Controller

We extend our general model of the TLS, described in Chapter 2, by adding a third entity which observes metrics attained by frameworks and influences the resource allocations made by the GS. We call this new entity the *feedback controller*. A schematic overview of the updated model is provided in Figure 4.4. The feedback controller monitors the frameworks in terms of their performance.

The feedback controller uses a policy to translate observed framework performances to a resource recommendation to be carried out by the GS. In the case of our TLS, the metric to be observed is the median job slowdown, and the resource recommendation is an update to the weight ratio of DRF. A detailed description of the policy, in the case of two frameworks, is as follows:

1. Every minute, compare the median job slowdown over the time window of the last x minutes for both frameworks, where x is a configurable experiment parameter.

2. If the median job slowdown between the frameworks differ more than a factor of 1.5, but not more than 3, adjust the allocation ratio such that the framework with a smaller job

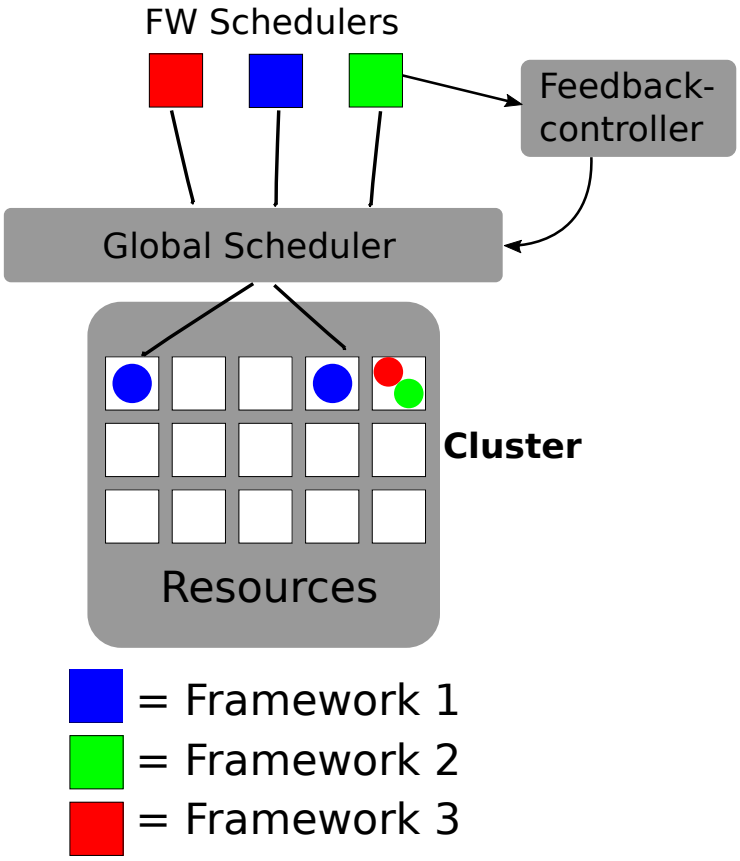


Figure 4.4: A schematic overview of the general TLS model with an added feedback controller

slowdown receives 5% less and the framework which has a larger job slowdown receives 5% more than its current allocation share.

3. If the median job slowdown between the frameworks differ more than a factor of 3, do the same as 2., but adjust by 10% instead of 5%.

4. At any point in time the adjusted ratio will not be more skewed than 1:8, which translates to roughly 11% and 89%.

In the next section we will use this policy on a similar experiment as in the previous section, without statically configuring an allocation ratio in Mesos.

Caveat In the following experiments using a feedback controller, we will use job slowdowns as our metric to quantify the performance of frameworks. In turn, we use the job slowdowns to make decisions which framework was to acquire more or less resources during the dynamic allocation. We realize that in a real setting, getting reference response times, as we have done in Section 4.2 is not always possible. However, in that case, the service times can be gotten

at runtime which can proxy the reference response times. For both approaches the requisite remains that jobs submitted can be classified to jobs ran in the past in order to calculate job slowdowns.

4.4.2 Using the Feedback Controller with Two Frameworks

In this experiment we use the extended TLS which includes our feedback controller. We run the same workloads of W_2 and W_4 with a combined imposed load of 0.7 for 10 worker nodes as in Section 4.3.2. We continue to use the configuration of attempt A2, in which we over-subscribe memory to allow proper CPU resource sharing among the frameworks. Our goal is to get a median job slowdowns for both frameworks which are closer to each other compared to the statically assigned allocation weights ratio of DRF. The idea is that by dynamically setting the allocation ratio during the experiment, we can mitigate the divergence of job slowdown performance of the frameworks during times of contention.

Because we do not know the optimal window size, we run multiple instances of the experiment by using a window size of 5, 10 and 20 minutes.

The job slowdown characteristics are in Table 4.9. Comparing the results to the static weights allocation ratios, in Table 4.7 for the 1:2 ratio and Table 4.8 for the 1:4 ratio, shows that we can achieve similar performance balance by using a feedback mechanism. The smallest difference between framework performance, as measured by mean and average job slowdown, is achieved by using the smaller window size of five minutes. We see a trend that the results for the 1:2 workload has smaller median and mean slowdowns overall compared to the 1:4 workload. This is unexpected as the total load in the system for both ratios is 0.7. However, at this point we cannot explain the trend.

4.4.3 Using the Feedback Controller with Three Frameworks

In order to further validate our approach to dynamically allocate resources to frameworks, we rerun a similar experiment, but with three frameworks instead of two. We initially ran the experiments on the DAS-4, but as we will see, due to memory constraints the service times inflate, and causes the system to be overloaded. Therefore, we also run the three-framework on the DAS-5, the newest generation of the DAS, which has more memory per node. In the next two paragraphs we will first explain how our policies can be extended to cope with more than two frameworks, and secondly, show and discuss the slowdown characteristics.

Extending Allocation Policies

To generalize our policy we extend it to cater to more than two frameworks. We extend the policy by ordering the frameworks in terms of their median job slowdown, and apply the original policy on the first and last framework in the sequence, then on the second and penultimate, and so on. Similar to the two-framework policy, we do not skew the ratio between frameworks more than 1:8. If the number of frameworks is odd, we disregard the framework in the middle of the sequence.

Job Slowdown Statistics with Dynamic Allocation					
Ratio	Window size	FW	Median	Mean	95th percentile
1:2	5 minutes	F_1	2.10	8.75	33.89
		F_2	2.62	10.11	43.32
	10 minutes	F_1	1.69	7.49	28.12
		F_2	2.32	8.80	38.48
	20 minutes	F_1	1.70	6.71	27.11
		F_2	2.49	11.84	56.90
1:4	5 minutes	F_1	6.47	32.00	161.34
		F_2	8.12	28.45	112.91
	10 minutes	F_1	3.61	29.51	162.51
		F_2	8.97	28.15	109.13
	20 minutes	F_1	3.76	26.46	159.98
		F_2	8.71	27.65	112.30

Table 4.9: Job slowdown statistics for experiments with dynamic allocation, using a feedback controller. Two workloads using different ratios for imposed load between frameworks are run on the system. For both workloads the values used for the window size for feedback control are 5, 10 and 20 minutes.

Unequal Workloads Characteristics for three Frameworks			
Workload	Imposed Load	Arrival Rate (s^{-1})	#Jobs
DAS-4 1:2:4 ratio			
$W'_{4,l}$	0.1	0.005 14	111
$W'_{4,m}$	0.2	0.010 28	222
$W'_{4,h}$	0.4	0.020 56	444
DAS-5 1:2:4 ratio			
$W'_{5,l}$	0.1	0.007 16	155
$W'_{5,m}$	0.2	0.014 32	309
$W'_{5,h}$	0.4	0.028 65	619

Table 4.10: Workload characteristics for two instances of the experiment with feedback controller and three frameworks. The total imposed load on a system of 15 nodes is 0.7

Experiment Setup

We extend our used experimental setup so far which exists of 10 physical nodes and add five nodes. The ratio of imposed loads of the frameworks is 1:2:4. The workload characteristics for the DAS-4 and DAS-5 can be seen in Table 4.10. To generate the 1:2:4 workloads for the DAS-5, we have rerun the reference experiment in order to get the service-times for the queries on the more modern hardware of the DAS-5.

Experiment Results and Discussion for DAS-4

The results of using three frameworks with a 1:2:4 ratio for imposed load are given in Table 4.11. We report the mean and average job slowdown for the three instances of the experiment, which differ on the window size of the feedback controller. In general, we can see that we do not attain a good performance balance.

These slowdowns are unexpected as we expected job slowdown statistics similar as attained with two frameworks. However, we found that the measured load for the workloads on the DAS-4 are approximately 30% larger than the imposed loads, i.e., the service times of individual queries are considerably larger than the service times attained during the reference experiment of Section 4.2. The significant increase in imposed load explains the larger median and mean slowdowns. Ideally our feedback controller should cope, even during this high imposed load, but the performances achieved between the framework with the lowest imposed load, and the framework with the highest load differ with a factor of 4. The measured load is approximately 0.9 and we've seen in Section 4.3.1 that the system cannot handle a load of 0.9. We exclude the experiment results from further discussion.

However, we investigated the reason for the unexpected load increase further. We found that Spark implicitly relies on the OS pagecache during the shuffle phases of a query. Especially for query 2C and 3C, which feature a relatively large shuffle phase, have significantly larger service

Job Slowdown Statistics - DAS-4						
Ratio	Window size	FW	Workload	Median	Mean	95th percentile
1:2:4	5 minutes	F_1	$W'_{4,l}$	8.95	25.29	98.08
		F_2	$W'_{4,m}$	18.92	37.96	114.42
		F_3	$W'_{4,h}$	37.30	52.96	150.02
	10 minutes	F_1	$W'_{4,l}$	5.03	19.81	75.98
		F_2	$W'_{4,m}$	17.68	38.62	116.40
		F_3	$W'_{4,h}$	42.36	54.14	158.08
	20 minutes	F_1	$W'_{4,l}$	5.03	18.33	65.71
		F_2	$W'_{4,m}$	13.98	39.14	139.92
		F_3	$W'_{4,h}$	20.12	40.84	126.05

Table 4.11: Job slowdown statistics for experiments using a feedback controller. Three workloads with different imposed loads in a ratio of 1:2:4 for a total load of 0.7 were run in the setup of the DAS-4. The experiment has been run for three instances, each using a different window size, 5, 10, and 20 minutes, for feedback control

times when three concurrent frameworks are running. The larger service times are result of the following . When running three frameworks concurrently on DAS-4 nodes, each framework runs a Spark executor on each Mesos slave, in total they use $15\,\mathrm{GiB}$ per slave. This leaves roughly $9\,\mathrm{GiB}$ of memory on the slaves for pagecache. The OS divides the pagecache over the three framework processes running on the slaves, resulting in $3\,\mathrm{GiB}$ of pagecache per framework. This is considerably less than the amount of pagecache, which was available for the single Spark framework during the reference experiment in Section 4.2.2.

Experiment Results and Discussion for DAS-5

Because we have difficulty running three frameworks on the DAS-4 because of memory constraints, we will run the same experiment on the DAS-5, which has more than double the amount of memory per node. However, since the DAS-5 is considerably faster than its predecessor, we rerun the reference experiment similar to Section 4.2 to get service times for the queries when run on the DAS-5. Using the reference service times, see Table 4.12, we calculate the mean average query service time using our job distribution defined in Section 4.2.2 as follows:

$$\begin{aligned} \overline{T_Q} &= 0.75 \cdot T_{small} &&+ 0.20 \cdot T_{medium} &&+ 0.05 \cdot T_{large} \\ &= 0.75 \cdot 255.4s &&+ 0.20 \cdot 4987.8s &&+ 0.05 \cdot 9726.2 \\ &= 1675.4s. \end{aligned} \tag{4.3}$$

With the mean query service time we calculate new workloads, $W'_{5,l}$, $W'_{5,m}$, and $W'_{5,h}$, and their characteristics are described in Table 4.10.

Reference Service Times on the DAS-5				
	Service Time (s)	**Response Time (s)**	**Size**	**Avg service time (s)**
Query 1				
A	231.4	7.99	Small	255.4
B	232.4	7.82		
C	302.4	9.65		
Query 2				
A	4243.8	131.1	Medium	4987.8
B	4716.9	143.9		
C	5161.2	156.9		
Query 3				
A	5315.1	166.1		
B	5501.8	170.3		
C	9726.2	278.7	Large	9726.2

Table 4.12: Service times and response times for the queries of the BDB when run on the DAS-5. The queries are classified in three classes: small, medium, and large, and classes small and medium have multiple types of queries.

The experiments results for running three frameworks concurrently on the DAS-5 are shown in Table 4.13. The results show job slowdowns which are close to each other for each workload. In the best case, with a window size of five minutes, there is a difference of $2\times$ between the framework with the smallest imposed load, F_1, and the framework with the largest imposed load, F_3. There seems to be a trade-off between performance balance for the median and 95th percentile job slowdowns on the one hand (5 minute window), or slightly worse performance balance and 95th percentile values, but lower median job slowdowns (20 minute window). For the 10 minute windows, we achieve a performance balance where the F_1 and F_3 differ with a factor of approximately $1.7\times$. This is close to the performance balances of 1.25 achieved for two frameworks in Table 4.9, and this is a considerable improvement to the results of running three frameworks on the DAS-4, where the performance balance differs by a factor of $4\times$.

4.5 Conclusion

In this chapter we evaluated the feasibility of achieving performance balance for multiple frameworks in a two-level scheduler, consisting of Spark on Mesos, for batch job workloads with different imposed loads. Using Spark on Mesos with two frameworks, which have different imposed loads, can result in mean and median job slowdowns which differ more than $10\times$ between

Job Slowdown Statistics - DAS-5					
Ratio	Window size	FW	Median	Mean	95th percentile
1:2:4	5 minutes	F_1	1.52	9.15	53.58
		F_2	1.90	13.28	76.28
		F_3	3.00	12.07	43.76
	10 minutes	F_1	1.60	13.90	102.68
		F_2	1.91	12.90	66.32
		F_3	2.74	11.45	48.63
	20 minutes	F_1	1.27	11.13	60.84
		F_2	2.16	17.48	106.75
		F_3	2.83	13.14	57.12

Table 4.13: Job slowdown results for experiments using a feedback controller. Three workloads using a 1:2:4 ratio for imposed load between frameworks are run on the system. For all workloads the values used for the window size for feedback control are 5, 10 and 20 minutes. These results are attained on the DAS-5.

the two frameworks. We show that by setting an allocation ratio in Mesos based on a priori knowledge of the workload characteristics, we can achieve job slowdowns for multiple frameworks which are well within a factor of two compared to each other.

The performance balance can also be gained without knowing the imposed load of workloads a priori, by using a feedback controller which continually observes framework performance during runtime and updates the allocation weight ratio of DRF dynamically.

Chapter 5

Conclusions and Future Work

Computing in distributed systems such as clusters and datacenters has become mainstream. Prices for hardware have decreased and access to datacenters has increased, due to fast internet connections and the shift to cloud computing. However, cluster and datacenter administrators are presented with the task to increase utilization of their hardware. A new generation of cluster schedulers have appeared, which add a layer of abstraction in the cluster to handle resources in a dynamic manner such that time varying workloads can be run more efficiently. Since these cluster schedulers are relatively new, it is unclear how they differ from each other. Furthermore, the policies for taking advantage of these cluster schedulers to dynamically allocate resources are not yet understood. Below we summarize the main conclusions of the research questions of this thesis in Section 5.1, followed by ideas for future work in Section 5.2.

5.1 Conclusions

We provide the conclusions of this thesis by stating the research questions of Section 1.2 and their corresponding conclusions.

RQ1 *How can we characterize and differentiate existing two-level-scheduler designs?*

We have defined a taxonomy of Two-level Schedulers (TLSs) to differentiate 6 state-of-the-art Global Schedulers (GSs) which are either open-source or described in the literature. The taxonomy describes the GSs using several aspects. One of the most decisive aspects is how TLSs can differ in the separation of responsibility between resource and task allocation. There are three prime examples of TLSs, which can represent the extremes on the spectrum of scheduler separation. In the middle of spectrum lies Mesos, which has a clear separation of responsibility for resource and task allocation. The Mesos master is responsible for allocating resources fairly to the frameworks, and the frameworks are responsible placing tasks on resources. Then, at one extreme of the spectrum lies the Quasar scheduler, where the GS decides both resource *and* task allocation. The frameworks running under Quasar do not have influence on which resources their tasks are being placed, this task allocation logic is inside the GS of Quasar. At the other extreme, there is the Omega cluster scheduler where frameworks schedule both resources and task allocation.

A single GS is not present in Omega and frameworks have to cooperate to allocate tasks on resources fairly.

The mechanisms how these responsibilities are separated can have implications. For example, on the extremes, where resource and task allocations are handled by either the GS or the frameworks, there is a *global view* of the cluster. With a global view better task placements can be made to allocate tasks of different frameworks, which do not interfere. This is contrast to schedulers like Mesos and YARN where there isn't a global view for task placement since the GS dictates which subset of the cluster resources are allocated to frameworks.

RQ2 *What is the performance of a single Spark framework under Mesos for interactive queries?*

We have defined an evaluation method to measure the performance of interactive queries on Spark-Mesos. The interactive queries are characterized by their short task times and the requirement that the result output is sent back to the end-user terminal, which implicates that there is extra step of network activity during the execution of the query.

In our specific case of Spark-Mesos there are three configuration parameters which have shown to be of interest: the Spark mode, the allocation interval of Mesos, and the partition size of Spark. The underlying problem of varying performance results for different combinations of the configuration parameters is under-utilization. When using the so-called fine-grained Spark mode, there is considerable overhead for each task in the query. By choosing a lower allocation interval and/or lower number of partitions, the overhead is reduced. In this regard Mesos is not a perfect abstraction for resource allocation by a framework, as Spark needs to know the implementation details of Mesos to perform well. Furthermore, we note that this is a direct implication of using *pushed resources* by Mesos, which is an aspect described in our model used to answer RQ1.

Depending on configuration parameters settings and type of query, the performance, measured as response time, can differ more than $5\times$ between the fastest and slowest execution. Although we can influence Mesos and Spark to handle interactive queries well, this does lead to values for the configuration parameters, which in turn lead to either high load for Mesos, or needs repartitioning of data which might not always be applicable. We conclude that Spark under Mesos currently does not handle short interactive queries well, without resorting to the so-called coarse-grained mode of Spark. However, in this mode we cannot share resources between multiple frameworks, which is required for our initial aim of achieving performance balance. Thus, we continue to investigate performance balance for *non*-interactive queries in RQ3.

RQ3 *Can we achieve a performance balance for non-interactive data-analytics queries between multiple Spark frameworks running under Mesos?*

We have evaluated the performance balance by using a experiment setup, which had two concurrently running Spark frameworks, with each their own independent workload. We have tested three strategies to evaluate the feasibility of performance balance. First, we naively let Mesos distribute resources fairly between frameworks. This works well when

the workloads of the two frameworks have workloads with equal imposed load. However, when the imposed loads are unequal the achieved performances for the workloads, expressed as median job slowdown, are not balanced, being unbalanced by a factor of 5.

Second, we evaluated the performance balance when two frameworks have workloads with an unequal imposed load. We tested two instances, an instance where two frameworks have workloads which differ in imposed load with a 1:2 ratio, and another instance where the workloads differ with a ratio of 1:4. We instruct Mesos to offer resources in the same ratio to the frameworks, as their imposed load on the system. This worked well, the performances achieved, expressed as median job slowdown for the frameworks, are within a factor 1.5 to each other. There were some practical problems achieving this performance balance for the 1:4 experiment instance, and we devised two solutions to solve this problem. One solution depends on changing configuration parameters of Mesos, the other solution is to change actual scheduling code of Mesos. However, in this strategy we assume we know the imposed load a priori, which might not be a viable assumption in a real deployment.

Third, we introduced a feedback controller to our TLS model which monitors performance metrics of multiple frameworks. This makes it possible to not rely on a priori knowledge of the imposed load of the workload, and react to online changes of performance metrics and accordingly instruct Mesos to change its offer policy dynamically. The performance balance was within a factor of 1.25 and 1.70, for the median job slowdown when using two frameworks and three frameworks, respectively.

This thesis has been executed in the anticipation that performance balance would be relatively easy to achieve. However, a simple scenario with one type of GS, Mesos, and one type of framework, Spark, has proven to be laborious. Understanding the interplay of configurations between Mesos and Spark is unpredictable and intractable as it approaches exponential search space when trying out combinations of configuration parameters. The official documentation of the systems is often lacking on details of implications for the parameters. Furthermore, Spark specifically has shown to be unpredictable in terms of performance in the situation when we tried to use three Spark frameworks. The underlying problem in this case was that Spark implicitly relies on the Operating System's page cache.

Our expectation for the field of cluster scheduling is that if the end-goal is to run existing frameworks without significant rewriting of their inner workings, the global scheduler will need to extend its framework interfaces to be rich to cater every use-case. If we follow the trends of Mesos and YARN, whose development can be followed very well, we can see functionality added which resembles functionality seen in monolithic classical schedulers.

5.2 Future Work

Although our work shows performance balance can be achieved, some issues during experimentation are not completely understood. We suggest the following direction for future work:

1. In Section 4.4.3 we found that running three frameworks was not possible due to increasing service times. However, we expected that this would be possible, since three frameworks should not exceed the resources of the cluster. Furthermore we found that the amount of Operating System pagecache is of importance for the performance achieved by Spark. A new reference experiment could be evaluated on the DAS-4, in which isolation mechanisms are used, e.g., the CGroups [28] subsystem of Linux, to get service times when Spark is limited to $5\,\mathrm{GiB}$ as enforced by CGroups. Using the new reference times, we would repeat the experiment with three frameworks, to see if using the above isolation mechanism results in more predictable performances, without increasing service times, when running more than two frameworks.

2. We were not able to run more types of frameworks besides Spark. However, to achieve high utilization in datacenters , we should investigate achieving performance balance for other instances of Data Analytics frameworks such as Hadoop, and other types of frameworks such as webserver applications and realtime frameworks. For the last two examples, a different method of interfacing to Mesos is expected to be needed. Because, for example for a webserver application it is difficult to uphold Service Level Agreements, which are on millisecond level, and starting a task through Mesos for every webrequest will not be sufficient and more elaborate schemes will have to be used which would still allow for dynamic resource sharing between multiple frameworks.

   Also it should be investigated how to run different types of frameworks concurrently, thus it should be investigated how heterogeneous performance metrics can be compared in order to achieve performance balance.

3. It should be investigated why our approach of offering resources to frameworks in the same ratio as their imposing load on the system (Section 4.3.3), results in higher mean and median job slowdowns for the 1:4 workload, compared to the 1:2 workload. The total load in both the 1:2 and 1:4 workload is 0.7, and we have used a simplified simulator, which uses formulas from *queuing theory*, to validate our hypothesis that we expect similar job slowdown statistics between the two workloads with different ratios. However, better simulators, specific to Spark, are necessary and should be implemented to further validate our approach of offering resources in the same ratio to frameworks as their imposed load.

Bibliography

[1] Apache hadoop. http://hadoop.apache.org/.

[2] Apache hadoop yarn âĂŞ concepts and applications - hortonworks. http://hortonworks.com/blog/apache-hadoop-yarn-concepts-and-applications/.

[3] Apache spark resource management and yarn app models. http://blog.cloudera.com/blog/2014/05/apache-spark-resource-management-and-yarn-app-models/.

[4] Apple details how it rebuilt siri on mesos. https://mesosphere.com/blog/2015/04/23/apple-details-j-a-r-v-i-s-the-mesos-framework-that-runs-siri/. Accessed: 13-07-2016.

[5] Big data benchmark. https://amplab.cs.berkeley.edu/benchmark/.

[6] databricks/spark-sql-perf. https://github.com/databricks/spark-sql-perf/blob/v0.1.0/src/main/scala/com/databricks/spark/sql/perf/bigdata/Queries.scala. Accessed: 2016-08-05.

[7] Giant data: Mapreduce and hadoop. http://www.admin-magazine.com/HPC/Articles/MapReduce-and-Hadoop. Accessed: 27-08-2016.

[8] Mesos clusters growing to monster sizes. http://www.nextplatform.com/2016/03/24/mesos-clusters-growing-monster-sizes/. Accessed: 13-07-2016.

[9] Sort benchmark. http://sortbenchmark.org/. Accessed: 19-07-2016.

[10] Sparrow. distributed low-latency scheduling. https://www.eecs.berkeley.edu/~keo/talks/sparrow-sosp-talk.pptx. Accessed: 29-07-2016.

[11] Henri Bal, Dick Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff. A medium-scale distributed system for computer science research: Infrastructure for the long term. *Computer*, 49(5):54–63, 2016.

[12] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proceedings of the VLDB Endowment*, 5(12):1802–1813, 2012.

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

[14] Pamela Delgado, Florin Dinu, Diego Didona, and Willy Zwaenepoel. Eagle: A better hybrid data center scheduler. Technical report, 2016.

[15] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 499–510, 2015.

[16] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

[17] Wolfgang Gentzsch. Sun grid engine: Towards creating a compute power grid. In *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 35–36. IEEE, 2001.

[18] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[19] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, volume 11, pages 24–24, 2011.

[20] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[21] Aleksandra Kuzmanovska, Rudolf H. Mak, and Dick Epema. KOALA-F: A resource manager for scheduling frameworks in clusters. In *Cluster, Cloud and Grid Computing (CCGRID), 16th IEEE/ACM Int'l Symp. on*, 2016.

[22] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[23] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.

[24] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J Abadi, David J DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 165–178. ACM, 2009.

[25] Zujie Ren, Xianghua Xu, Jian Wan, Weisong Shi, and Min Zhou. Workload characterization on a production hadoop cluster: A case study on taobao. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pages 3–13. IEEE, 2012.

[26] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364. ACM, 2013.

[27] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Özcan. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proceedings of the VLDB Endowment*, 8(13):2110–2121, 2015.

[28] Balbir Singh and Vaidyanathan Srinivasan. Containers: Challenges with the memory resource controller and its performance. In *Ottawa Linux Symposium (OLS)*, page 209. Citeseer, 2007.

[29] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency and computation: practice and experience*, 17(2-4):323–356, 2005.

[30] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[31] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[32] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 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*, pages 2–2. USENIX Association, 2012.

[33] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

Appendix A

Big Data Benchmark Queries

In this appendix we enumerate the SQL queries of the *Big Data Benchmark* used in Chapter 3 and Chapter 4. There are three different classes of queries, each having three variants.

```
SELECT
  pageURL,
  pageRank
FROM rankings
  WHERE
  pageRank > 1000
```

Listing 1: Query 1a

```
SELECT
  pageURL,
  pageRank
FROM rankings
WHERE
  pageRank > 100
```

Listing 2: Query 1b

```
SELECT
  pageURL,
  pageRank
FROM rankings
WHERE
  pageRank > 10
```

Listing 3: Query 1c

```
SELECT
  SUBSTR(sourceIP, 1, 8),
  SUM(adRevenue)
FROM uservisits
GROUP BY
  SUBSTR(sourceIP, 1, 8)
```

Listing 4: Query 2a

```
SELECT
  SUBSTR(sourceIP, 1, 10),
  SUM(adRevenue)
FROM uservisits
GROUP BY
  SUBSTR(sourceIP, 1, 10)
```

Listing 5: Query 2b

```
SELECT
  SUBSTR(sourceIP, 1, 12),
  SUM(adRevenue)
FROM uservisits
GROUP BY
  SUBSTR(sourceIP, 1, 12)
```

Listing 6: Query 2c

```
SELECT sourceIP, totalRevenue, avgPageRank
FROM
  (SELECT sourceIP,
    AVG(pageRank) as avgPageRank,
    SUM(adRevenue) as totalRevenue
  FROM Rankings AS R, UserVisits AS UV
  WHERE R.pageURL = UV.destURL
    AND UV.visitDate > "1980-01-01"
    AND UV.visitDate < "1980-04-01"
  GROUP BY UV.sourceIP) tmp
ORDER BY totalRevenue DESC LIMIT 1
```

Listing 7: Query 3a

```
SELECT sourceIP, totalRevenue, avgPageRank
FROM
  (SELECT sourceIP,
    AVG(pageRank) as avgPageRank,
    SUM(adRevenue) as totalRevenue
  FROM Rankings AS R, UserVisits AS UV
  WHERE R.pageURL = UV.destURL
    AND UV.visitDate > "1980-01-01"
    AND UV.visitDate < "1983-01-01"
  GROUP BY UV.sourceIP) tmp
ORDER BY totalRevenue DESC LIMIT 1
```

Listing 8: Query 3b

```
SELECT sourceIP, totalRevenue, avgPageRank
FROM
  (SELECT sourceIP,
    AVG(pageRank) as avgPageRank,
    SUM(adRevenue) as totalRevenue
  FROM Rankings AS R, UserVisits AS UV
  WHERE R.pageURL = UV.destURL
    AND UV.visitDate > "1980-01-01"
    AND UV.visitDate < "2010-01-01"
  GROUP BY UV.sourceIP) tmp
ORDER BY totalRevenue DESC LIMIT 1
```

Listing 9: Query 3c