Studying Recommender Systems to Enhance Distributed Computing Schedulers

by

Henri Maxime Demoulin

Department of Computer Science
Duke University

Date: ____________________

Approved:

____
Benjamin C. Lee, Supervisor

____
Jeffrey Chase

____
Bruce MacDowell Maggs

Thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science in the Department of Computer Science
in the Graduate School of Duke University
2016

Abstract

Studying Recommender Systems to Enhance Distributed Computing Schedulers

by

Henri Maxime Demoulin

Department of Computer Science
Duke University

Date: ____________________
Approved:

____
Benjamin C. Lee, Supervisor

____
Jeffrey Chase

____
Bruce MacDowell Maggs

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science in the Graduate School of Duke University
2016

Copyright © 2016 by Henri Maxime Demoulin
All rights reserved except the rights granted by the
Creative Commons Attribution-Noncommercial Licence

Abstract

Distributed Computing frameworks belong to a class of programming models that allow developers to launch workloads on large clusters of machines. Due to the dramatic increase in the volume of data gathered by ubiquitous computing devices, data analytic workloads have become a common case among distributed computing applications, making Data Science an entire field of Computer Science. We argue that Data Scientist's concern lays in three main components: a dataset, a sequence of operations they wish to apply on this dataset, and some constraint they may have related to their work (performances, QoS, budget, etc). However, it is actually extremely difficult, without domain expertise, to perform data science. One need to select the right amount and type of resources, pick up a framework, and configure it. Also, users are often running their application in shared environments, ruled by schedulers expecting them to specify precisely their resource needs. Inherent to the distributed and concurrent nature of the cited frameworks, monitoring and profiling are hard, high dimensional problems that block users from making the right configuration choices and determining the right amount of resources they need. Paradoxically, the system is gathering a large amount of monitoring data at runtime, which remains unused.

In the ideal abstraction we envision for data scientists, the system is adaptive, able to exploit monitoring data to learn about workloads, and process user requests into a tailored execution context. In this work, we study different techniques that have

been used to make steps toward such system awareness, and explore a new way to do so by implementing machine learning techniques to recommend a specific subset of system configurations for Apache Spark applications. Furthermore, we present an in depth study of Apache Spark executors configuration, which highlight the complexity in choosing the best one for a given workload.

A la beauté des rêves, à l'amour qui nous tient.

Contents

Abstract **iv**

List of Tables **x**

List of Figures **xi**

Acknowledgements **xii**

Introduction **1**

1 Background and related work **7**

1.1 Distributed Computing Frameworks 8

1.1.1 MPI 8

1.1.2 Hadoop 9

1.1.3 Spark 10

1.2 Distributed Computing Schedulers 11

1.2.1 Borg 12

1.2.2 Mesos 13

1.2.3 Omega 14

1.2.4 YARN 15

1.2.5 Octopus 16

1.2.6 Reflections 17

1.3 Performance measurements and profiling in Distributed Systems . . . 18

1.4 Other attempts to optimize distributed computing setups 19

1.4.1 Google . . . 19
1.4.2 Paragon . . . 19
1.4.3 Starfish . . . 19
1.5 Recommender Systems . . . 20
1.5.1 Content Based recommenders . . . 20
1.5.2 Collaborative Filtering . . . 21
1.6 Spark survey . . . 23
2 Constraint driven resource requests 25
3 A recommender system for Apache Spark 28
3.1 What to recommend for? . . . 29
3.2 Determining the initial configuration space . . . 29
3.3 picking a model . . . 30
3.4 validating the model . . . 31
3.4.1 Testbed . . . 31
3.4.2 Workloads . . . 31
3.4.3 Training set . . . 32
3.4.4 Validation metrics . . . 32
3.4.5 Validation results . . . 33
3.5 Playing Scenarios . . . 37
3.5.1 Scenario 1 . . . 38
3.5.2 Scenario 2 . . . 39
3.5.3 Scenario 3 . . . 41
3.5.4 Scenario 4 . . . 42
3.6 Debrief . . . 43
4 Daemon architecture for a recommender system 44

5 Case studies: impact of executor size in Apache Spark **46**

6 Conclusion **65**

7 Appendices **67**

.1 Compute ranking score for a set of predictions 68

.2 Exhaustive description of our configuration spaces 68

List of Tables

3.1 Scenario 1 PageRank 38
3.2 Scenario 1 Wordcount 39
3.3 Scenario 1 Connected Components 39
3.4 Scenario 2 PageRank 40
3.5 Scenario 2 WordCount 40
3.6 Scenario 2 Connected Components 40
3.7 Scenario 3 PageRank 41
3.8 Scenario 3 WordCount 41
3.9 Scenario 3 Connected Component 42
3.10 Scenario 4 PageRank 42
3.11 Scenario 4 WordCount 42
3.12 Scenario 4 Connected Components 43
5.1 Configuration analyzed 64
1 Appendix: Configuration spaces 69

List of Figures

1 A new abstraction for Data Scientists 4

3.1 Accuracy: space 112 UBCF 34

3.2 Accuracy: space 112 IBCF 35

3.3 Accuracy: space 26 UBCF 36

3.4 Accuracy: space 26 IBCF 37

4.1 Implementation of the RS 45

5.1 Impact of executor configurations on Wordcount 49

5.2 Wordcount details for 58-4-4. Blue dots represents tasks' start, green tasks' end 50

5.3 Wordcount details for 232-1-1. Blue dots represents tasks' start, green tasks' end 52

5.4 Drivers heap usage. Blue dots represents tasks' start, green tasks' end 53

5.5 Impact of executor configurations on Connected Components 55

5.6 CC details for 58-4-4. Blue dots represents tasks' start, green tasks' end 57

5.7 CC details for 232-1-1. Blue dots represents tasks' start, green tasks' end 58

5.8 Impact of executor configurations on Pagerank 60

5.9 Maximum heap usage for executors across configurations for Pagerank 61

5.10 Average cpu load for executors across configurations for Pagerank . . 62

5.11 Top and worst configurations performance for Wordcount 64

Acknowledgements

While not very impressive in my eyes, this work allowed me to discover academic research and make all kinds of mistakes I hope to avoid during my upcoming Ph.D. I must thank my advisor, Benjamin C. Lee, as well as all the professors of the Duke Computer Science department who always provided me with their attention when I needed it, namely Jeff Chase, Bruce Maggs, Theo Benson, and Landon Cox. I enjoy writing a special mention to Xiaobai Sun, who has inspired me a lot during my Master Degree, Bruce Donald, and Marilyn Buttler whom acted as a mother to me during the past two years. The whole body of Duke's Computer Science faculty was also very responsive to any question I ever had.

To my friends Brandon and Brendan, thank you for all the good times spent together, as well as all to the other grad students of the department whom I enjoyed the company of. Also, thank you Julia for not having been there yet and leaving me some free time to work on this thesis. Those two years of research wouldn't have been as fulfilling without James and the Cobo Brothers Dance Company.

Lastly, my acknowledgments go to my parents, for without their unconditional love, this adventure would never have been (obviously), and to my friends abroad, for letting me abandon everything I had in France to pursue my quest here, while providing me with their support.

Introduction

Datacenters and Distributed Computing

This thesis is written in a time where computer software operate on unprecedentely seen large datasets, and compute not from one machine but hundreds. Many services used by millions, sometime billions of people, are hosted into dedicated warehouse called datacenters. Those facilities are built around computers, providing an environment where power supplies are highly reliable, cooling a first class citizen, and every piece of hardware is designed for space efficiency and heat concerns. In those environments, the chance that a single component fails is very high, an maintenance teams roam all day to replace hard drives, network cards, memory modules, network cables, routing devices, and so on.

Many efforts have been made to create software platforms providing abstractions to manage a large number of computers. Tools such as OpenStack, Dell Crowbar, Chef and Puppet, Windows SystemCenter, VMware vCenter, Microsoft Azure or Amazon AWS, can be used to deploy, configure, and maintain a large number of computers, making them act all together as single compute units, called clusters.

Among the software operating on such clusters, one can name e-commerce websites, social networks and search engines; databases; online streaming services, such as video and music providers; and so on. A special class of workloads running in datacenters are distributed computing applications. Those applications implement one or several programming model which allow to process in parallel a large amount

of data. There exists various degrees of abstraction to do so: some of them leave a lot of responsibilities to the programmer, while some other aims to provide a new layer of abstraction, where developers do not have to be wary about the underlying infrastructure. The latter have been a huge step in making large datasets and compute resources accessible to non technician scientists and users, empowering the phenomenon known as Big Data.

It is very likely that the amount of data we gather and store, as a society, will keep increasing. While distributed computing frameworks are already providing a neat abstraction to users, they are still in their infancy. We believe that much needs to be done to make those abstractions smarter, able to automatically provide tailored context of execution to their users, and see many places where optimization can be done.

Defining a new abstraction for Data Scientists

The abstraction we envision for data scientists and users of Big Data in general, is able to not only hide the details of the underlying infrastructure, but also to automatically adapt to their workload and constraints. Users should be able to solely have to bother about the data they want to operate on, the sequence of operations they want to apply on those data, and a set of constraints which are likely to impact the execution of their computation.

Actually, frameworks such as Apache Spark and Apache Hadoop (thanks to its rich ecosystem), are providing a way to declare a pipeline of operations on a dataset. However, users still have to acquire a large domain expertise to be able to operate those frameworks on the underlying system, as they still have to select the right mix of resources and configure the framework. As, due to the end of Moore's Law and Dennard's scaling, hardware heterogeneity is likely to increase in the future, that burden will become increasingly complex, and be a brake for users, even if the

technology provides them ways to do their computation. In addition, as we will detail later, most schedulers expect users to clearly define their resource vectors, a daunting task for non system-experts. Users tend to over estimate their resource demand, a behavior that is punished by many schedulers and hinders user experience. As we see in the next section, profiling and monitoring distributed systems in general is a difficult problem from which we want to take users away.

We present a very high level view of the abstraction we envision in figure 1. First, by one way or another, users craft a request made of the three components we mentioned earlier, and forward it to the system. Then, the system, analyzing that request, is able to pick the right mix of hardware, compute framework, and system configuration, based on user's request and scheduling policies enforced by the system main's scheduler. The environment we picture can be managed by a shared state, two level, or monolithic cluster. Such a system could offer automation for users, but also let them control by themselves the tuning, while offering suggestions.

In this thesis, we survey solutions that have been proposed or implemented to optimize such data intensive environments. Very often, those techniques rely on a dataset of monitored data, which allow the system to categorize applications and users based on profiling. It is then possible to make inferences about what factors made a program run successfully, and what not. The idea is to have the system reason about what it has experienced in the past, and formalize the process of analyzing workload behavior. Our own contribution to the topic is the study of recommender systems to proceed with such analysis.

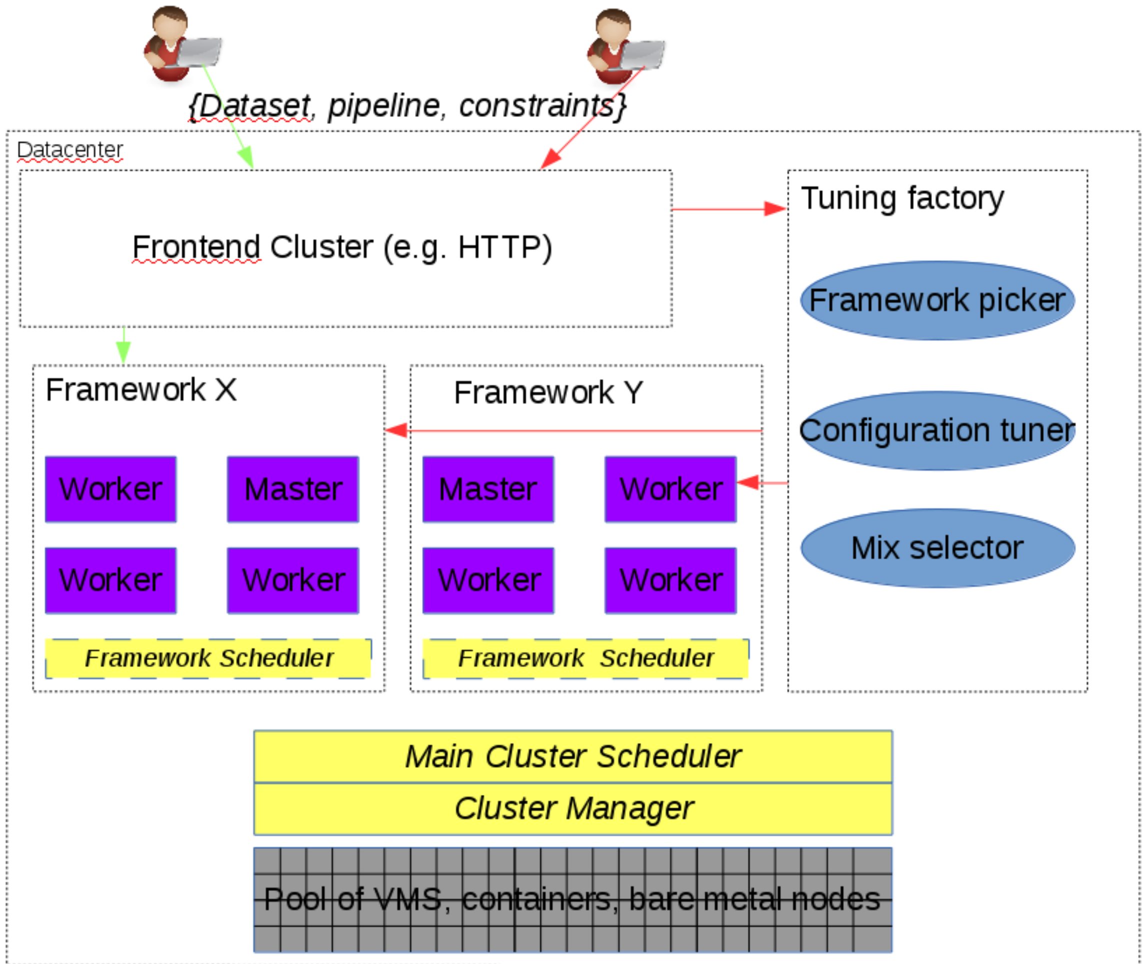


FIGURE 1: A new abstraction for Data Scientists

Challenges in reasoning about distributed computing system performance

It has proven extremely difficult to develop a sense of what could go wrong in distributed systems. Failures can be physical (hardware failures) or human (configuration, bug in the software) or a combination of circumstances, like an unexpected corner case which appeared due to a shift in usage patterns. Not only is it difficult to reason about thousands of nodes running the same program together, it becomes even more complicated to do so when several software are interacting together into a complex pipeline of operations, in a tiered ecosystem. From "basic" considerations such as time, consistency, ordering of operations, to more advanced situations where

a workload is decoupled in multiple stages, with intermediary results flowing through the system from one stage to another, the number of factors impacting the end result is tremendous.

Monitoring the behaviour of a computer implies understanding the fundamental of operating systems and architecture: why is some piece of hardware more suited to a given operation, why are some memory layers slower than others, how can two processes compete for hardware resources, why is it more expensive to access memory from a remote NUMA domain, why has the amount of memory allocated to a process an impact on garbage collection, why processes are CPU, memory, or I/O intensive, how can a distributed computation, in appearance embarrassingly parallel, still suffer from unexpected bottlenecks, etc.

The dimensions of those questions get even higher when we start adding machines to the system: what is the impact of the network, how well are the caching tiers and storage bays performing, how does the scheduler handle several applications in the cluster, what is the impact of hardware heterogeneity? What amount of shuffling data does an application generates, how to partition a dataset to increase the level of parallelism in the cluster and optimize the performances of individual processes, what are the constraints imposed by sharing policies in a multi tenant cluster, etc.

At the framework level, many parameters can have an impact on applications' life cycle, ranging from the size of the executing processes to the compression algorithm used to transfer data from and to main memory. The actual modus operandi appears to be tuning frameworks empirically, based on what seemed to work for previous runs.

As we take more parameters into consideration, the number of dimensions augment, as so many combinations of system and framework combinations are possible. Parameters can be conflicting, often implying performance and efficiency trades off, which makes it hard to pick the right ones. We find that very often, users lack the expertise required to finely tune their framework to carry at best the workload they

wish to compute, and lack, in their entourage, the presence of experienced people who can help them in making such decisions.

Contributions of this thesis

Our contributions are the following:

1. We briefly discuss techniques to measure Distributed System performances,
2. We introduce the notion of constraints driven resource request,
3. We apply collaborative filtering and recommender systems to Apache Spark to guide users in configuring their application, and study the viability of such methods,
4. We discuss representative Apache Spark workloads under different executor configurations,
5. We provide a software which allow anyone to exploit Spark workloads traces to visualize performances and get recommendations.

1

Background and related work

1.1 Distributed Computing Frameworks

The goal of a Distributed Computing framework is to provide an abstraction to developers which allows them to use a large amount of resources while seeing them as an unified pool. The framework will provide a protocol to handle data and task distribution on the nodes, as well as fault tolerance concerns, like relaunching a task which failed (whether it comes from the task or from the node). The framework does not know the underlying topology and must cope with infrastructure evolution. Popular frameworks such as MapReduce and Spark also provide a programming model which allows to reason about a large dataset as a unified piece (location transparency), and to logically split it in a multitude of components independently processed by the cluster's nodes. The dataset itself is often stored on a distributed file system, on the very same nodes where tasks are processed; developers can control its partitioning, and create a computation pipeline to be applied on each part.

1.1.1 MPI

The Message Passing Interface is a language independent standard allowing programs to communicate with other elements in the system. MPI provides a fundamental building block in distributed computing systems, namely the message passing protocol which allow programs to access the distributed memory in a computer or in remote machines. However, it requires a user to implement message passing in her code, where higher level frameworks such as Spark and Hadoop handle that transparently. The consequence is that users have to know much more about the underlying infrastructure, distracting them from their program. MPI is more often than not categorized as a High Performance Computing (HPC) framework, rather than a Distributed Computing one, however, it shares many properties with the latter.

1.1.2 Hadoop

Apache Hadoop is the open source implementation of the MapReduce paper [13] , first published in 2004 at OSDI. The project was started in 2006 by Doug Cutting and Mike Cafarella, employed by Yahoo! at the time. The first official release of the Apache distribution was shipped in 2011.

Hadoop is made of three main building blocks: the distributed file system, HDFS (an open source implementation of the Google File System [17]), the MapReduce API, and YARN [33], the scheduler.

The MapReduce programming model comes from functional programming: a user can apply (map) a function to a set of independent data segments, and then aggregate the intermediate results (reduce). A classic example is an implementation of WordCount, where the user wants to know the number of occurrences of every word in a corpus. First the text is split in a certain number of partitions, which are each mapped to a function which operates local word count in its assigned partition. Then, once map() is done, all the separate intermediate results are joined and merged in a reduce() phase.

Typically, Hadoop's implementation of Map Reduce stores the data in HDFS, which ensure that a sufficient amount of *chunks* are available and spread throughout the cluster. YARN implements a technique called delay scheduling [35], which tries to maximize data locality to enhance performances by reducing the cost of network I/O. We discuss further data locality and delay scheduling later in the chapter.

HDFS architecture is a master/slave paradigm, where the master keeps all the information about data placement, and authorization, whereas the slaves store *chunks* of data and report status to the master regularly. To ensure fault tolerance and availability, each slave has a specified number of replicas (3 by default), which are located in different physical locations to strengthen their resilience to a physical failure. To

ensure consistency, each set of replicas has a primary, which validates (or deny) a sequence of *mutations* on a piece of data. Enforcing a set of mutations makes the data *defined*, which means that the version is acknowledged as being the definitive one.

1.1.3 Spark

Apache Spark [37] belongs to a new generation of Distributed Computing frameworks. A key motivation behind Spark was to enhance iterative workloads performances by performing in memory computations. The raw Hadoop implementation is quite inefficient for those workloads, mainly due to the numerous access to disk required to carry the application to its end. Spark made two main contributions that tackle this issue: first it introduce a new type of data structure, called Resilient Distributed Dataset [36], which plays an major role in enabling in memory computing; second, it proposes a rich software architecture that ease the creation of Distributed Computing programs for a wide range of category, such as graph processing, machine learning and streaming.

RDDs are immutable collections of data, which the user can create from an initial storage (HDFS, remote object store, local file, etc), and modify through *transformations*. At some point, the user can collect the new data generated by such transformations, by *collection* methods. RDD allows the implementation of a refined version of the MapReduce model, enriching the variety of possible user defined functions, and allowing them to create a real computation pipeline on a dataset. RDDs are computed lazily, which means that only when a user wants to collect the data, will transformations be applied. Each RDD has a lineage, meaning that the sequence of operations applied to the initial slice of a dataset is known and recorded. Tasks in Spark are operations on RDDs. Each operates on a partition of an RDD, and the relationship between tasks is either concurrent or dependent. The nature of tasks

(transformation or collection) defines two type of dependencies: *narrow* and *wide*. A narrow dependency is typically the result of two consecutive map() functions on an RDD. A wide dependency often induces a shuffle, meaning that intermediate data must be moved through the cluster to be reduced().

At compile time, Spark analyzes operations applied to RDDs in the program, and generates a Directed Acyclic Graph. It tries to pack as many narrow dependencies as possible in one stage, and typically places both ends of a wide dependency into different stages. Tasks are preferably assigned to workers holding the required data, with a technique called delay scheduling. It allows a task to decline a worker's offer if it doesn't hold the data required. Different locality levels are configured, node, rack, and cluster locality, allowing Spark to gradually increase a task's tolerance to non locality. The framework can carry a computation to its end even in the event of discrete failures by replaying RDD's lineage.

RDDs can be instantiated in specialized types depending on the type of computation they store data for. For example, there exist graph RDDs, file RDDs, and columnar stores RDDs.

1.2 Distributed Computing Schedulers

There are three main categories of distributed systems schedulers: monolithic, two levels, and shared states schedulers. The first type has a global view of the resource pool, and must implement a variety of possibly conflicting allocation policies for every framework in the cluster. As the amount of specialized frameworks and applications increase, such monolithism is hard to maintain and can become a bottleneck. Two level schedulers, such as Mesos, allocate a pool of resources to a framework, which then implement its own specialized policies at will. Each framework has a partial, locked view of the global resource pool. Hadoop, for example, can use YARN with such a subset to manage concurrent Map Reduce applications. Lastly, shared state

schedulers offer a global view of the resource to a set of specialized schedulers. They typically reach a higher level of consolidation, and have a great flexibility in terms of differences in the allocation policies, but must carefully manage changes in the global state to mitigate conflicting allocation decisions.

1.2.1 Borg

More than a distributed computing scheduler, Borg [34] not only manage jobs and tasks allocation on a tremendous amount of machines at Google, it also manage their life cycle by setting up operating systems and installing software components, ensures fault tolerance by replacing failing nodes, and provide a whole monitoring ecosystem. In addition, it provides simulators which allow to test new versions of the system, new scheduling policies, and so on, on a subset of the resource, before an actual deployment in production. It also provides a specialized language for users to declare their workload.

Borg operates on *cells*, which are clusters containing about 10k heterogeneous nodes, defined by the network fabric which interconnects them. Each cell hosts long lived services and short lived batch jobs. *Borgmasters* have total control over the cell, and each node has a *Borglet* agent. Initially monolithic, Borgmasters can be replicated (in a Paxos style quorum) to support specialized schedulers accessing a shared state of the cell (inspired by Omega, which we describe later, and is also a Google project).

Borg acts as the kernel of a cell, scheduling tasks on nodes from pending, running, and dead queues. It can prioritize tasks and preempt them to free up resources. It reduces the need for fairness policies by defining resource quotas for each user/application. Those users can define preferences (which we identify as constraints driven resource requests in our terminology), which Borg takes into account. Users are still responsible for declaring the amount of resources they desire for an appli-

cation, and have great incentives to be precise about the amount of resources they request: Borg's managed ecosystem is priced, and users *purchase* resource quotas.

Also, as Borg collects a tremendous amount of monitoring data and logs, it is able to perform a lot of internal optimizations, which are prioritized over individual users' preferences (for the larger good). In addition, it can empirically determine application preferred settings for some users, thought the method used to make such predictions are not disclosed in the paper.

Borg is similar in essence to cluster managers and schedulers used by other, equally sized internet giants: Microsoft's Autopilot [22], Facebook's Tupperware [5], and Twitter's Aurora [4].

1.2.2 Mesos

Mesos is a two level modular scheduler for distributed computing frameworks. The modular design allows it to implement different resource allocation policies. Frameworks implement they own schedulers, which then take responsibility for resource management of their allocated pool.

Mesos aims to be scalable and fault tolerant. It minimize its role in the cluster by simply offering resources from the global pool to frameworks, which can in turn accept or reject those offers. Mesos allocation modules can implement weighed fairness, prioritization, task preemption, and so on. It provides isolation to co-existing frameworks by running containers [7] on workers, where tasks are executed. Mesos' master is replicated through Zookeeper [8].

The philosophy behind Mesos is, as advocated by Omega and later implemented in Borg, that a monolithic scheduler responsible for providing every possible allocation policies is not maintainable, nor is it scalable. Pushing those features to the frameworks, with a set of "efficiency" incentives, such as a revocation time on workers, and accounting individual allocation time as occupancy from the framework,

makes it in practice much easier to consolidate the cluster with multiple, different frameworks. In a scenario where a large number of users want to use the cluster (and where its operators want to maximize utilization and efficiency) and run dramatically different frameworks, or different versions of the same framework, such a design makes a lot of sense. There is a small risk that a specific mix of frameworks leads to perpetual resource contention, and an overall lower cluster utilization than with statically partitioned shares, but in such cases it might suffice to either statically allocate a partition of the cluster to one of the contending framework, or to switch to a shared state scheduler. Mesos' paper also mention that such a conflicting combination can lead to resource fragmentation, which results in sub-optimal usage of the cluster.

We note that Apache Spark was initially designed to be a specialized scheduler and framework running on Mesos, to showcase its efficiency.

1.2.3 Omega

Omega is a Google project aiming to tackle the challenges raised by Borg's monolithism, which was a bottleneck that made it difficult to scale and hard to update with new scheduling policies.

Omega also steps away from the two level resource pools advocated by schedulers such as Mesos, where each framework is given a share of the cluster to use at will. Instead, it allows each framework's scheduler to access the global state of the cluster, and run its allocation algorithm based on a user defined resource request and the current state of the cluster. In addition, it is possible to let the framework opportunistically claim more than the initial resource request. When a framework scheduler tries to update the cluster state, potential conflicts have to be resolved as the view of the cluster obtained by the framework while starting the allocation might have changed when the algorithm came to an end. A framework updates the

global state in an atomic transaction. If the transaction fails, the framework has to run its scheduling algorithm again, with an updated view of the cluster returned by the transaction.

That mode of operation fits very well environments where the nature of workloads is heterogeneous, with, for example, coexisting long lived services, and short lived batch jobs. It also allows schedulers to implement independent, specialized policies. We note that for contexts where only a handful of frameworks are sharing a cluster, the two level scheduling might be easier to manage out of the box, and that it provides a degree of isolation for jobs not provided by a Shared state scheduler.

1.2.4 YARN

YARN is Hadoop's scheduler. It was conceived to replace a monolithic implementation of Hadoop, where the programming model and scheduler were an unique block. Its goal is to maximize utilization, while providing reliability and high availability to jobs. Another prized feature of YARN is the support of multiple programming model. Frameworks such as Spark can use it to manage resource allocation. YARN is built of three main components. The Resource Manager (RM), ubiquitous overseer of a global resource pool, can distribute them to users under the form of leases. A lease on a node allow to spawn a process, named container, which can execute tasks. A container has access to a limited amount of memory and cores (CPU time) on the worker node. Each node runs a Node Manager (NM), which interacts with the RM to report its state and let the RM gather a global view of the resource pool through its interactions with each node's NM. Once an application is accepted and can run on the cluster, an Application Master (AM) is run on one of the allocated nodes. AM will be responsible for coordinating application's tasks. For example, in the case of Spark, AM will run the application's driver (in "cluster" mode).

In YARN, users can express to the RM, a ResourceRequest, through the AM,

which specifies a number of containers, their size (memory and cores) and locality preferences. In the paper words, users are expected to "express their needs clearly".

YARN implements a fair scheduler (HFS [1]), which maximize the share of memory applications can get out of the cluster. Resources can be arranged in pools, which support a hierarchical relationship. HFS can be extended by a technique named Dominant Resource Fairness [18], which maximizes the minimum share of a resource for an application, as well as guarantee some desirable properties such as envy-freeness, strategy-proofness, and gives sharing incentives to users. DRF process users' ResourceRequest to determine their share.

1.2.5 Octopus

Octopus [29] is a specialized scheduler for Graphlab [25], a distributed computing framework for graph analytics. It tackles the special challenges encountered to schedule concurrently multiple graph processing applications from different users. It does so by applying different placement policies such as First Fit and FIFO with Round Robing Filling, as well as a fair scheduler. Octopus does not preempt running tasks, so might lead to poor performances in certain cases where a particularly long job has to wait for two long ones to finish, for example.

Very relevant to the present thesis are some highlights offered by Octopus' paper:

1. Demonstration of an "elbow point" where throwing more resources at the system results in reduced performances,

2. Mention to the importance and difficulty to size executor processes. We quote the paper: *"Also, in case of GraphX, it is non-trivial to know the optimal executor memory size and number of cores desirable by a job."*,

3. Mention that while it is possible to predict and recommend such configurations parameters, as in YARN, Borg, and other schedulers, users are still responsible

for declaring their resource vectors. Again, we quote the paper: *"Machine learning algorithms can be used to predict the optimal number of nodes required for a job. But currently our scheduler takes this as an input from the user."*

1.2.6 Reflections

Across the various scheduler we have presented, a common factor is that they all push toward the user the responsibility of knowing how many resources their application need, as well as running time configurations such as executor size. It is indeed arguable that schedulers themselves should not implement the logic of determining such vectors for users, as it would result in an increase in complexity, equivalent to the one perceived in monolithic schedulers when trying to fit every framework's specialized allocation policies. However, there is clearly a gap to fill between schedulers and users, as the amount of factors impacting their decision is extremely difficult to think about. Some study reveal how inaccurate users estimation can be when requesting resources [10], which supports our motivations.

1.3 Performance measurements and profiling in Distributed Systems

Distributed Systems have been under profiler's attention as soon as the first of them were running. Different approaches have been taken, from reactive techniques such as MAPE (Monitor, Analyze, Plan, execute), to more complex modeling [16]. Modeling, in despite of the difficulty of picking a right model, has proved to be a powerful technique which allow operators to make predictions about future executions of similar workloads. Building a model can be made with data that is going to be there anyway, and/or by adding instrumentation to gather more specific ones.

Techniques such as blocked analysis [28] have been developed to identify the impact of a specific component on the overall application behavior. Blocked analysis project the run of an application by simulating infinite bandwidth or memory, and analyzing at which point such abundance could benefit the workload, in order to reveal true bottlenecks.

A whole class of study has been dedicated to stress specific components of a distributed application, leading to the development of micro benchmark suites such as iBench [14]. Such analysis allow to detect workloads' sensibilities, which in turn helps to measure interference they can cause to each other. This is particularly important in a setting where we try to consolidate applications on the same machines to increase efficiency.

Efforts such as [27] have been focusing on developing a notion of *knowledge plane*, which convey system's information and allow to model the system as a more manipulable entity.

Lastly, we mention systems such as Pivot [26], which propose a rich query language to not only leverage dynamic instrumentation but also apply *causal tracing* techniques to capture more precisely the succession of events in a system which led to a given state.

1.4 Other attempts to optimize distributed computing setups

We mention some other notable attempts to help user configuring their framework. Those are only drops in an ocean of research papers.

1.4.1 Google

The Borg paper mention Google's attempt to automatize low priority users. Quoting the paper: *"Our solution has been to build automation tools and services that run on top of Borg, and determine appropriate settings from experimentation".*

1.4.2 Paragon

Our work is greatly inspired by Paragon [15], which uses collaborative filtering with SVD to find a best hardware match for a workload, taking into account hardware heterogeneity and contention for resources. Our system has similar motivations and tries to find the best software configuration for a workload, using IBCF or UBCF recommenders. Ultimately it seems likely that an hybrid model could prevail.

1.4.3 Starfish

Starfish [21] is an optimization system built on top of Apache Hadoop [13]. It uses a profiler to capture workloads' performances, and provides a "what-if" engine to predict an application's behavior with different environment parameters. Our work has similar goals, but targets any distributed computing framework. Also, we present a study of collaborative filtering as a way to infer workloads' configurations for any valued metric, instead of building an application profile to do so.

1.5 Recommender Systems

Recommender systems are a class of machine learning tools that can provide information about user's preferences in a given context. Some traditional example includes online bookstores which would like to recommend new readings to users, airplane ticket brokers willing to advertise targeted flight promotions, or online music players promoting new songs. The idea is to match users' preferences to product characteristics.

Quite different mechanisms can be used to build a recommender system, the nature and availability of user's information being critical to their capabilities and efficiency. Some system might have *explicit* feedback from users, while some others only have access to *implicit* ones. We define by explicit feedback a voluntary action from a user, like rating a song, while an implicit feedback would be the fact that a user listened to that song. Explicit feedback often give a sense of appreciation (a high rating being interpreted as enjoyment from the user), while implicit ones are much harder to interpret.

The two main techniques in the field are Content Based and Collaborative Filtering recommenders.

1.5.1 Content Based recommenders

A content based recommender system is based on the information one can gather to characterize users and products. For example, we might define a user as having a gender, an age, a marital status, tastes for pop music and martial arts, and so on; equally, we could define a movie has being an action movie, themed with pop music, and starring a famous - but now deceased - Chinese practitioner of KungFu. The goal of the content based recommender system is to match attributes from the movie (which we will therefore refer to as *product*) and the user interest for those

attributes.

Content based recommenders are usually built of a content analyzer which defines users and products profiles, and a filtering component which match users to products. The system represents attributes in the form of vectors, and compute similarity by methods such as the cosine similarity or Hamming distance.

The reason why content based recommenders are difficult to craft for complex use case is that many parameters of the system are difficult to gather: how to select the right attributes for a product? What about attributes for fresh users who just entered the system? Also, should we have an human perform the gathering, or can it be delegated to a machine? All those questions pave the way for Collaborative Filtering, which tries to cope with all those uncertainties by automatically discovering patterns in the available data.

1.5.2 Collaborative Filtering

Collaborative filtering is an ensemble of techniques used in machine learning to fill missing entries in user-product matrices in order to build recommender systems. Contrarily to content based filtering, which relies on exhaustive characterization of users and products, collaborative filtering use past user behaviours to identify relationships between them. It has the benefit of not being bound to any particular context, but is suffering from the cold start problem, meaning that when system disposes of very few data, it is hardly accurate. The term has been coined by Goldberg et al., authors of what is arguably the first recommendation system, Tapestry [19].

Two main approaches to collaborative filtering are neighbor based models and latent factor models. The former forms clusters of items or users, identifying a cluster by computing some similarity measure on the existing ratings. In our scenario, a product is a system configuration, and a user an application an its parameter space.

Latent factor models are a third method which can be viewed as an automated

way to characterize users and products, similar to the profile built by content based filtering methods. Actors' factors can be represented in the form of vectors (for a product, a vector of factors indicate the possession of a factor, for a user, its interest in a set of factors), and a dot product on the "right" vectors can lead to a recommendation. However, as it can be hard to make a mapping between users and product factors, complex matrix factorization techniques such as SVD, CUR, or ALS, are employed.

1.6 Spark survey

We study a representative paper where people use Spark to conduct experiments, and illustrate how the configuration parameters used when benchmarking applications can be difficult to get. Many other papers we read did not specify any configuration, which can be interpreted either as convenient omission, either as a proof that the required expertise to choose a suitable configuration is an obstacle for the authors.

The studied paper [30], is a study of the memory utilization of analytic queries running on the Spark SQL engine. The point is to show that the framework underutilize the system capacity, which in turn allow for disaggregation, a technique which consist in provisioning separately various resources in a system, such as CPU and memory. Disagreggation makes it easier to scale a specific resource and thus reach the desired infrastructure composition for a workload.

The testbed used is made of 5 nodes, 4 workers and a master. Each worker has a single executor, consuming all of the (dual threaded) 8 cores, and 18 of the 24 GB available on the node. In their workload, a thread inside an executor's JVM correspond to a query. The evaluation keeps the number of executors fixed, but changes the number of threads (cores) it uses. The authors made many specific tuning to Spark to improve its memory efficiency, which might explain how they gained the expertise required to clearly detail their system configuration. Among the decision they made, we note that they disable shuffle spill, and redirect intermediate data to a tmpfs (in memory) partition. We might ask why they took that decision instead of giving more memory to executors (6GB were still available in the system): if an executor gets out of memory to hold shuffle data, why wouldn't the tmpfs partition not happen to be in the same situation?

Observing their result, what we see is that they seem to underutilize executor's memory, for the maximum batch they load in the heap is equal to $100000 * 128$B

= 12.8GB. We also note that the maximum bandwidth they measure is peaking at 1.9GB/s, with 16 threads, while they only reach 1.25KB/s for the same amount of data and 8 threads processing it. We ask the question: would they have doubled that 1.25KB/s throughput if they had used two 8 cores executors, with 9GB of memory and half the data each (i.e. loading $50000 * 128B$ = 6.4GB in memory)? The theoretical bandwidth would then have been 2.5GB/s, nothing that challenge their point (that system's bandwidth is underutilized), but still a 25% increase of the peak bandwidth. Also, the same reasoning could be applied to their result concerning the data throughput.

2

Constraint driven resource requests

With regard to the variety of users and contexts they work in, we advocate for constraints driven resource requests. A user might be willing to pay more to have a computation carried faster, but could also accept a significant cutoff in price at the cost of some performance reduction. Those constraints do not dictate scheduler's policy, nor require any change at their level: constraints determine the way users express their resource vectors. Being able to reason in terms of constraints pave the way for further automation, where the underlying infrastructure offers to a user a set of resources based on her preferences.

We define a few possible constraints users might want to meet. Many other exist, based on whatever indicator can be monitored from the system.

- *Energy efficiency*: Energy relates to cost in many ways, and we would like to carry a computation in a fashion where we obtain maximum performances for minimal energy consumption. While computers are not energy proportionals [12], some hardware are more efficient than others for specific workloads [11].

- *Memory efficiency*: What fraction of time is a workload spending in garbage collection, with regard to the running time? Are there configurations where that fraction is lower than other? What impact does it have on other indicators such as running time? Is the maximum heap usage in a resource container close the amount of memory available to that container? We believe that such questions have a subtle impact on each process, which reverberates throughout the cluster.

- *CPU efficiency*: What fraction of time was spent doing actual computation, versus the amount of blocked time? How close to the allocated CPU share is an application actual CPU utilization? Is the cost of compressing data outweighing the final benefits?

- *Cost efficiency*: Whatever our main constraint is, users might want to put it into perspective with the cost incurred in meeting that constraint. For example, by taking the ratio of dollar to running time, we might conclude that a slightly smaller resource vector will cost much less money and thus be preferable.

- *Geolocalisation*: Application might be preferably launched at a given location, possible a specific subset of racks in the datacenter, or in a specific datacenter where energy is cheaper, or where the legislation about data is more flexible.

Going further, we want to allow users to create complex sets of constraints. Doing so will allow them to assemble a set of factors which they can prioritize: if the scheduler can answer their requests as closely as possible, we expect it will push the quality of users' experience to a level yet unseen in traditional infrastructures.

We note that a major obstacle to constraint driven resource requests is the complexity to extract monitoring values and craft the right indicator representing at best a constraint.

3

A recommender system for Apache Spark

3.1 What to recommend for?

As we have seen earlier, there is a variety of factors affecting the performances of a distributed computing workload. We decide to narrow the scope and identify three of them, which are steadily configurable by users: the number of executor process, and their allocated cores and memory. The size of a process has indeed an impact on performance, as our in depth studies will reveal later, for reasons such as heap management, amount and frequency of data shuffled, shared variables between threads, etc. We designate an executor configuration by a code name made of those three factors, and therefore refer to them as such: e-m-c, where e is the number of executor processes, m and c the amount of memory and number of cores, respectively, allocated to *each* process.

3.2 Determining the initial configuration space

The amount of possible configurations we can generate is tied to the overall amount of resources in the share we have control on. In our testbed, we have 29 workers provided with 14GB of RAM and 8 cores, which yield a total set of about 3200 possible homogeneous combinations. Among those, many are unbalanced configurations, featuring executors with 14GB of RAM and 1 core, or 1GB of RAM and 8 cores, which under utilize the share.

We trim that space to retain two sets of configurations, one of size 26 and the other on size 112. Appendix 1 details both sets. Set **a**, of size 26 is a "realistic" set where each configuration matches a given ratio of memory to cores, and for each ratio, retain the ones which maximize executor's bin packing on our cluster. We select the following ratios: 4:1, 3:1, 2:1, 1:1, 3:2. Set **b** aims to create a space with more possibilities for recommendations.

We suspect that no single configuration is a best performer for every application,

so being able to determine the most suitable one will help the system in providing a better user experience. Lastly, some resource scheduler such as DRF assume that a user has one and only one preferred resource vector, which makes the scheduler strategy proof. If a user is now able to specify multiple satisfying vectors, that assumption is challenged.

3.3 picking a model

In order to get accurate recommendations, we need to select an algorithm that fits well with Spark use cases. We also need to define a configuration space which will not be too sparse at the beginning, but still could be extended in the future to get more precise recommendations.

We expect users to split their share in a natural way, meaning that they will use configurations multiples of a resource vector which tend to reach 100% of the worker allocated memory. For example, if our nodes have 14GB of memory and 8 cores, we expect users to create executors with $< 3.5, 2 >$, $< 7, 4 >$, and $< 14, 8 >$. Of course, every other vector splitting the original amount is possible and should be envisaged. If we are able to determine that a workload doesn't need to exploit the whole worker's allocation, it is possible to feedback that information to the cluster scheduler, allowing a user to free resources, and maybe to save some form of credit or currency for later.

Based on such usage assumption, we imagine that we will have ratings for very few configurations from applications, with regard to the magnitude of the configuration space. This means that, either memory based, model based, and latent factors models will not be able to find similarities for unrated products. One solution to cope with that behavior will be to select a smaller configuration space, made of "expected" vectors - such as the space **a** we described earlier - and extend it over time. Such extension is outside the scope of this work, but could be explored by picking, over

time, configurations outside the space used for the recommender, and performing a serie of workload sampling against them. Once we gather enough sample to meet the model's sparsity tolerance, we can train it again.

Thinking about the properties of the model, we propose various choices for the system. A memory based model will operate under the assumption that if an applications performed well on a configuration, a similar application is also likely to do well on it. We study later in this chapter a scenario where we use a memory based model. A model based algorithm, which assumes that if an application performed well on a configuration, it is likely to perform well on a similar one, seems intuitively much more correct. However, such models, retaining only a amount k of similar products, are known to lose in accuracy when sparsity augments, as shown in the next chapter. We try both of those models and study their usability in our scenarios.

3.4 validating the model

3.4.1 Testbed

We run our workloads on a testbed of 29 blades, all in the same rack. The blades are HP made, with 8 Xeon L5420 @ 2.50GHz cores and 16GB of DDR2 RAM. Spark Standalone 1.5.0 is running, with Hadoop 2.6.0 providing HDFS. Overall we have at our disposal 232 CPUs and 406GB of DRAM (we limit to 14GB the memory available to executors).

3.4.2 Workloads

We run the following workloads on our testbed:

- *WordCount*: We use a set of web pages released by Wikipedia, of 44 and 88 GB. WordCount is representative of CPU intensive applications, where the map phase computes a partial count of words, and the reduce phase aggregate those counts.

- *K-means*: A machine learning algorithm popular in data mining. It has been widely used for feature learning and cluster analysis, both important method in machine learning to characterize a dataset used to train a model. We use Apache Flink [3] to generate datasets of different size, using 4 and 5 data points. K-means is representative of CPU-bound workloads (being an iterative process over the data) as well as I/O bound ones (for the clustering part).

- *Connected Components*: this graph algorithm aims to find connected components in a directed graph. We use a graph representing the web, made available by Google in 2002 [6], composed of a roughly 900k nodes and more than 5m edges.

- *PageRank*: very popular graph algorithm ignited by Larry Page, aiming to rank web pages for their popularity, which is determined by the chance that a random walker ends up on the page and the rank of connected pages. We run it on the same Google graph, shrank to 1 million edges. PageRank is CPU bound.

3.4.3 Training set

We experiment validating IBCF and UBFC models with the configuration space we defined in the previous section and our selection of workloads. First we detail the performances of the model with a set of 6 applications (we run Wordcount and K-means with two different input size). Our objective is to predict configurations for running time. We normalize the scores per application on a 0 to 1 scale.

3.4.4 Validation metrics

We measure recommendations' accuracy by computing the Root Mean Square Error and a custom ranking score. We iteratively measure accuracy against sparsity in the

training set to estimate how much data we need to gather before expecting accurate recommendations.

The ranking score, which we describe in appendix .1, is focusing on ranking accuracy, and punish the model for serious mistakes in rankings. We define a serious mistake as being a product predicted to outperform another, when in reality it is not, and when the predicted performance is further than 10 or 15% lower than the one of the product it is doing prejudice to. A score close to 1 denotes accuracy, while at 0.5, users might be better off picking a configuration randomly than trusting the system. At the contrary, a low RMSE indicates accurate predictions. The Bellkor solution to the Netflix Grand Prize [24] announce an RMSE of 0.8870 for indicators ranging from 0 to 5. On our 0 to 1 scale, it would be equivalent to 0.1774, so we think of an RMSE of 0.17 as being a reasonable accuracy threshold for the recommender system.

3.4.5 Validation results

Figure 3.1 shows accuracy's evolution for the 112 configuration space and 6 applications, using UBCF. Being 10% tolerant theoretically allows us to run the recommendation system with only 50% of the data, while 15% tolerance allows us to support about 70% sparsity.

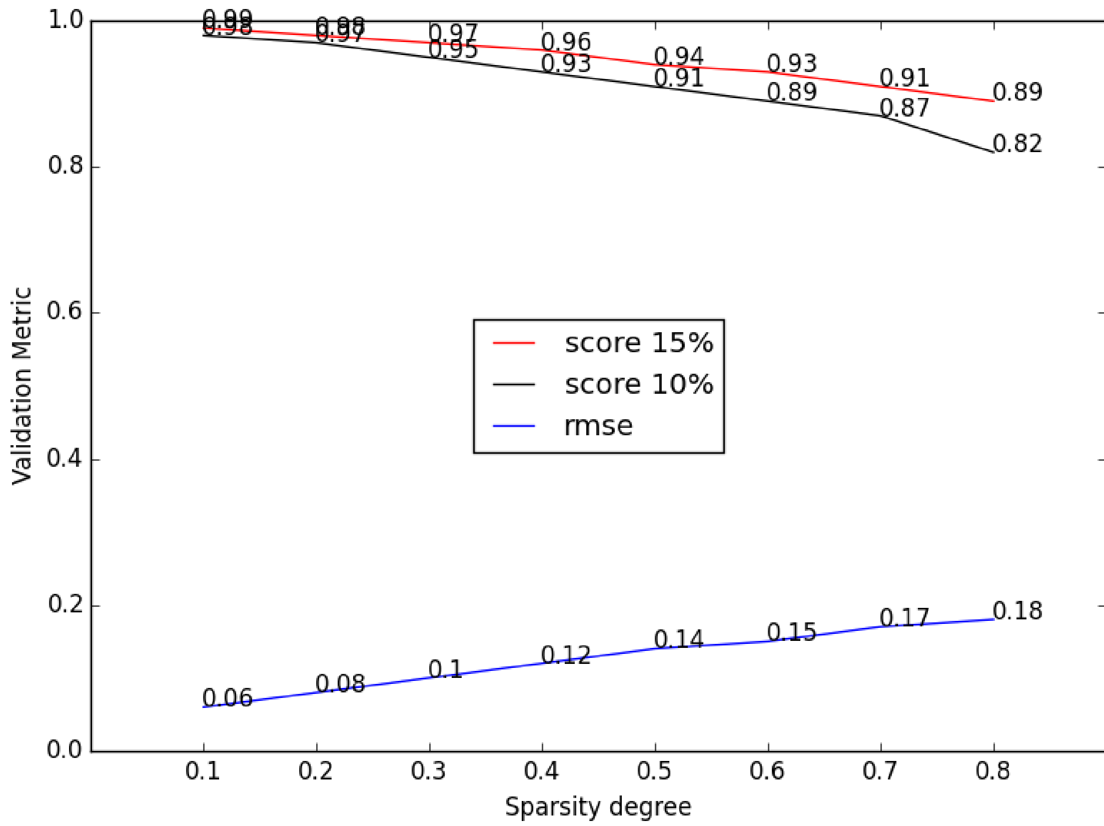


FIGURE 3.1: Accuracy: space 112 UBCF

Figure 3.2 shows that the same space with IBCF is much less tolerant: we do not calculate RMSE past 40% sparsity as the model is failing to come up with predictions for the missing entries.

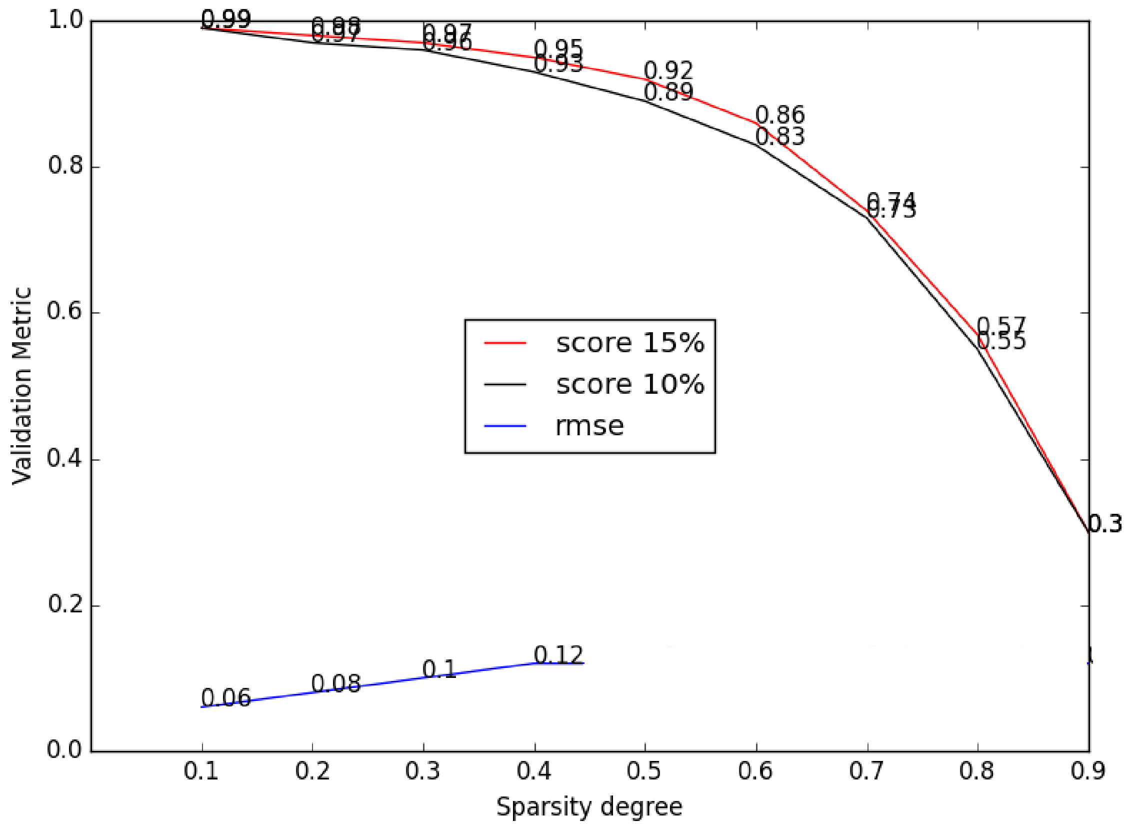


FIGURE 3.2: Accuracy: space 112 IBCF

Figure 3.3 and 3.4 repeat the validation for the 26 configuration space. UBCF seems to be consistently more accurate than IBCF, and we also note that having a smaller configuration space does not help the models operating under sparsity, which makes sense has both of them are relying upon groups of similar users/items to derive predictions. The more difficult it is to find those clusters, the more likely it is that the model won't be able to predict.

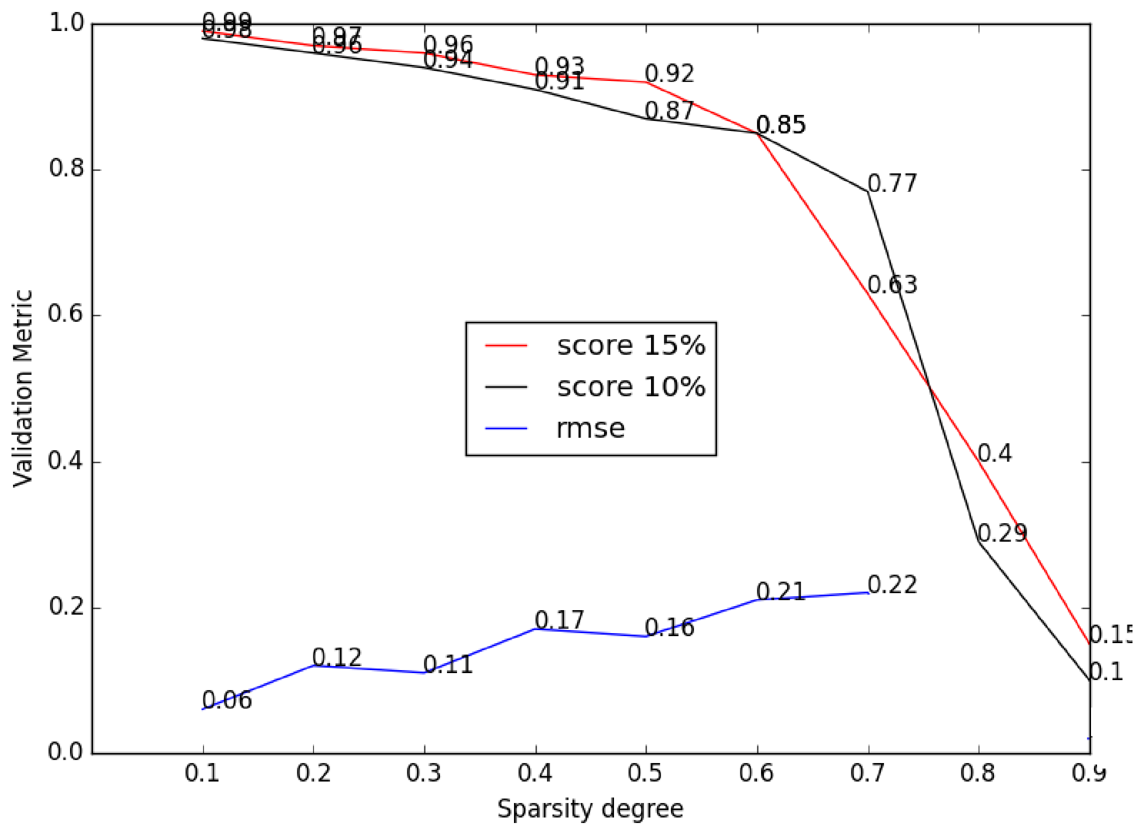


FIGURE 3.3: Accuracy: space 26 UBCF

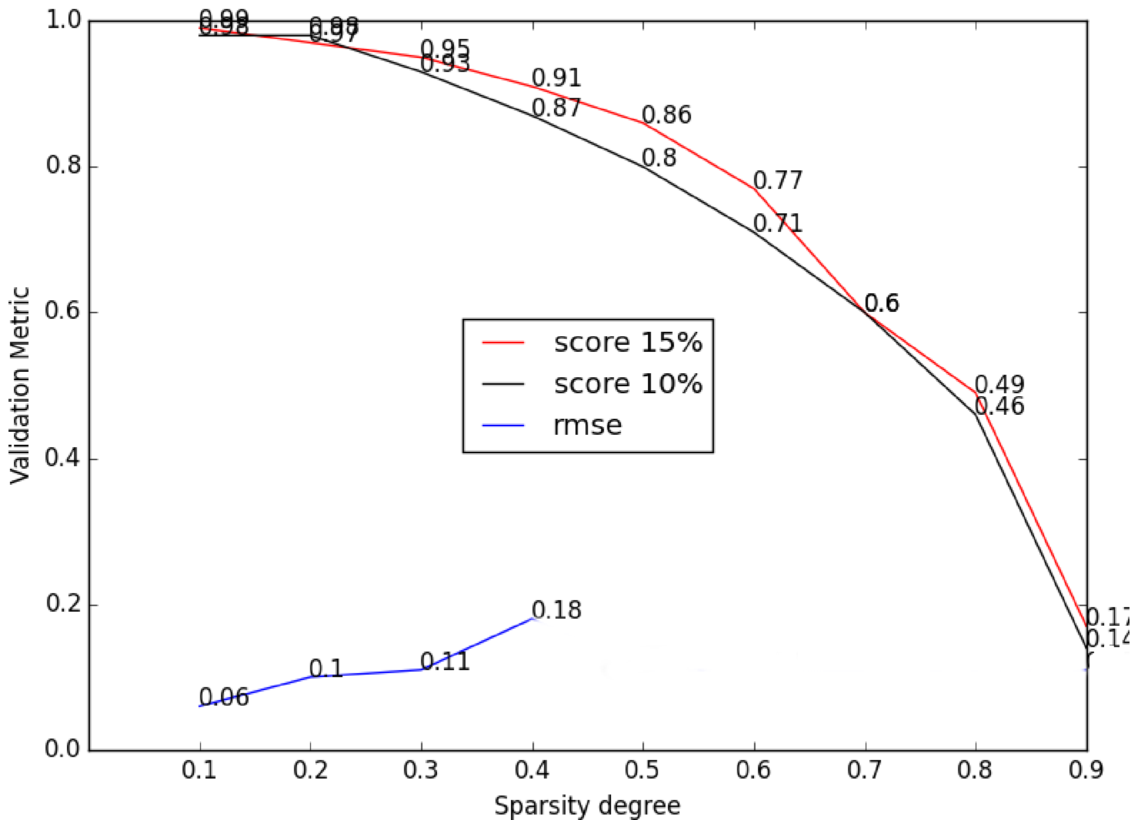


FIGURE 3.4: Accuracy: space 26 IBCF

3.5 Playing Scenarios

We present a sequence of scenarios where we evaluate the system. We presume that we, a group of users, have been allocated a share of resources in a cluster, and want to run Spark applications on it. We also presume in our scenarios that the recommender's database has already been warmed up by either previous users in our group or automatic runs of workloads picked from the set of application we wish to run on the cluster. We recognize, but do not address, the cold start problem that the system is facing when users just received their share of resources and haven't run anything yet. Previous works [38, 31, 32, 9] have been extensively reviewing and looking to solve or mitigate that problem.

We unroll the following scenarios:

1. In the 26 configuration space, 5 of 6 applications have been fully monitored. we train an UBCF model with those data, and then run the 6th application on the top 5 best overall performers observed for the 5 previous applications. Then, we run the recommender and examine the output. We present that experiment with Pagerank, Wordcount, and Connected Components.

2. We replay scenario 1 within the 112 configuration space.

3. We replay scenario 1 with IBCF

4. We replay scenario 2 with IBCF

Those scenario are all exhibiting a cold start problem for a new application entering the system.

3.5.1 Scenario 1

We play the scenario with Pagerank. Table 3.1 displays the top 5 best performers for the previously run applications, alongside the top 5 items proposed by the recommender. We observe that no configuration was predicted being better than our two initial top performers, but when we try the three next ones, we find that 87-4-2 is beating the best time (we also realize that we were recommended a significantly slower configuration, 29-12-6).

Table 3.1: Scenario 1 PageRank

Initial top 5	Runtime	Predicted top 5	Real runtimes
29-9-6	52	29-9-6	52
29-6-6	52	29-6-6	52
29-8-8	53	29-12-6	72
29-8-4	54	87-4-2	50
29-5-5	54	29-12-8	52

Table 3.2 and 3.3 presents the experience with Wordcount and Connected Components, respectively. In the case of Wordcount, this time, we were not able to identify better configurations thanks to the recommender system. If we totally benchmarked the 26 configuration space, we would have discovered that 29-12-6 was the optimal configuration, with a running time of 103 seconds.

Table 3.2: Scenario 1 Wordcount

Initial top 5	Runtime	Predicted top 5	Real runtimes
29-9-6	110	29-9-6	110
29-8-4	115	29-8-4	115
29-7-7	117	232-1-1	161
29-6-6	120	203-2-1	142
29-5-5	128	116-2-2	129

For Connected Components, we also could not find a better configurations by exploring the proposed ones. It turns out that one of the initially proposed configuration, 29-5-5, was the best performer across the whole space for CC.

Table 3.3: Scenario 1 Connected Components

Initial top 5	Runtime	Predicted top 5	Real runtimes
29-5-5	42	29-5-5	42
29-6-6	44	29-9-3	44
29-8-4	44	29-8-2	43
29-8-8	44	58-3-3	47
29-7-7	47	29-9-6	51

3.5.2 Scenario 2

We repeat scenario 1 with the larger configuration space **b**. We also assume that previously five applications have been entirely sampled, and try to get recommendations for a sixth one. We get the top 5 overall best performers from our database, run them, and trigger a new iteration of the system to get new configurations to explore.

While we don't get to discover a better configuration for Pagerank (49s being the actual best running time in this space, fact that we, ignorant users, can't know), we uncover two better performers for Wordcount, as well as a very slightly better one for CC.

Some behaviours are unveiled by the system: wordcount seems to abhor small executors in favor to bigger ones (from 232-1-1 to 29-5-7). But also, even if it is a CPU bound workload, it still benefits higher memory allocation, as we can from the increase from 29-5-7 to 29-10-6. We study in more details those three executions in the next section.

Table 3.4: Scenario 2 PageRank

Initial top 5	Runtime	Predicted top 5	Real runtimes
29-5-7	49	29-5-7	49
29-4-5	50	29-4-5	50
29-2-8	53	29-10-4	52
29-5-5	54	58-5-1	53
29-3-7	56	29-8-5	56

Table 3.5: Scenario 2 WordCount

Initial top 5	Runtime	Predicted top 5	Real runtimes
29-5-7	118	29-5-7	118
29-3-6	122	29-3-6	122
29-4-5	127	232-1-1	161
29-5-5	128	87-4-2	107
29-2-8	137	29-10-6	104

Table 3.6: Scenario 2 Connected Components

Initial top 5	Runtime	Predicted top 5	Real runtimes
29-5-7	43	29-5-7	43
29-5-6	44	29-5-6	44
29-3-6	45	29-10-6	46
29-4-8	45	87-4-2	49
29-4-6	45	29-10-4	42

3.5.3 Scenario 3

We replay scenario 1, but this time train the model using item based collaborative filtering. We chose the amount of items per cluster (k) to be 6.

Getting recommendations for Pagerank does not allow us to find better configurations, but we notice that past a certain amount of memory per executors, performances decreases dramatically.

Table 3.7: Scenario 3 PageRank

Initial top 5	Runtime	Predicted top 5	Real runtimes
29-9-6	52	29-6-6	52
29-6-6	52	29-9-6	52
29-8-8	53	29-12-4	72
29-8-4	54	29-14-7	82
29-5-5	54	29-12-8	52

Meanwhile, running the system for Wordcount allowed us to discover to better configurations again. The trends seems to be in favor of a relatively linear increase in performance by amount of resource thrown to the system.

Table 3.8: Scenario 3 WordCount

Initial top 5	Runtime	Predicted top 5	Real runtimes
29-9-6	110	29-9-6	110
29-8-4	115	58-4-4	118
29-7-7	117	29-8-4	115
29-6-6	120	29-12-8	104
29-5-5	128	29-10-5	108

Unfortunately, we were not able to discover any better configuration for CC.

Table 3.9: Scenario 3 Connected Component

Initial top 5	Runtime	Predicted top 5	Real runtimes
29-5-5	42	29-5-5	42
29-6-6	44	29-9-3	44
29-8-4	44	29-8-2	43
29-8-8	44	29-6-6	44
29-7-7	47	29-8-4	44

3.5.4 Scenario 4

In this scenario, we replay the number 2 with IBCF, keeping the number of items in a cluster of similar items to 6. The results are not really satisfying as we are only able to discover one better configuration for Wordcount.

Table 3.10: Scenario 4 PageRank

Initial top 5	Runtime	Predicted top 5	Real runtimes
29-5-7	49	29-5-7	49
29-4-5	50	29-5-6	49
29-2-8	53	29-6-6	53
29-5-5	54	29-4-5	50
29-3-7	56	29-9-8	53

Table 3.11: Scenario 4 WordCount

Initial top 5	Runtime	Predicted top 5	Real runtimes
29-5-7	118	29-5-7	118
29-3-6	122	29-6-6	120
29-4-5	127	29-7-8	121
29-5-5	128	29-5-6	112
29-2-8	137	29-3-8	133

Table 3.12: Scenario 4 Connected Components

Initial top 5	Runtime	Predicted top 5	Real runtimes
29-5-7	43	29-5-7	43
29-5-6	44	29-4-7	47
29-3-6	45	29-5-6	44
29-4-8	45	29-6-6	44
29-4-6	45	29-7-7	47

3.6 Debrief

The predictions are not really accurate for the scenarios emulating a cold start problem, in both our configuration space, for UCBF and IBCF. IBCF handles very poorly sparsity, and won't be able to make predictions when it is too high. However, running such models is very inexpensive and can help users exploring new configurations. Also, doing so reveals trends in workload performances under varying resource vectors.

4

Daemon architecture for a recommender system

The system we implement is made of three components: a submit wrapper which receive user's command and run spark-submit with every option we need to generate logs; a log crawler, and the recommender system, which can be configured to recommend for one or many constraints.

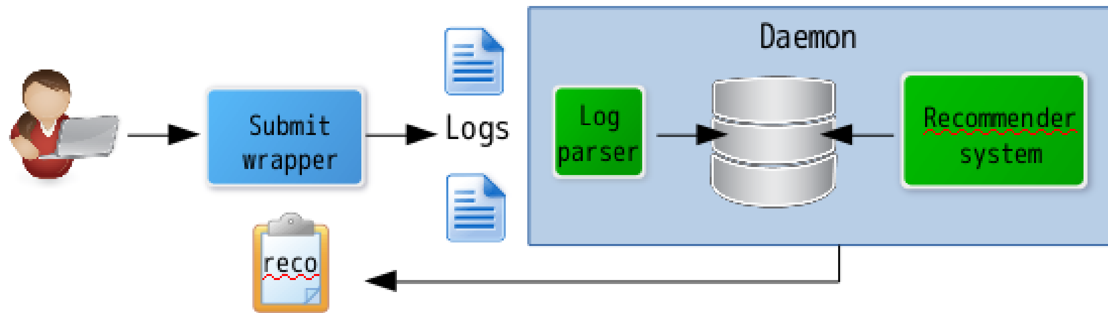


FIGURE 4.1: Implementation of the RS

The daemon runs periodically and update its recommendation, gaining in accuracy with time as the database growth. Users can consult results from a simple file and try new top recommendations. We ultimately envision automatic executor configurations guided by the system.

5

Case studies: impact of executor size in Apache Spark

In this section we provide an in depth study of Spark's behavior for different workloads under some of the configurations we detailed earlier. Specifically we:

1. Study the impact of different configurations for the same amount of resources.

2. Explain what makes some configuration top or worst performers

We proceed to the above analysis with Pagerank, Wordcount and Connected Components. First we look at the global behavior of each application under different configuration, then we look into more details to an application. We then repeat that study across applications to compare them.

Two takeaways from the previous chapter are that different configurations seems to be performing similarly, while they have different amount of resources available and resource partitioning, while some other perform really bad; and that a fleet of smaller executors performs globally worse than a small amount of bigger ones. What we are looking at is the reason why there is a sweet spot for a metric (here running time), given a configuration.

First we study a case where we split an equal amount of resources into different configurations, then we analyze traces for the two best and worst Wordcount configurations. In addition to running time, we analyze executor's heap usage, CPU load (as a measure of CPU time consumed by the executor's threads. We capture that information through JAVA's OSMXBean getProcessCpuLoad() method [2] and trigger the call within our BTrace probe every 0ms), and throughput, the amount of successful tasks per second completed by an application.

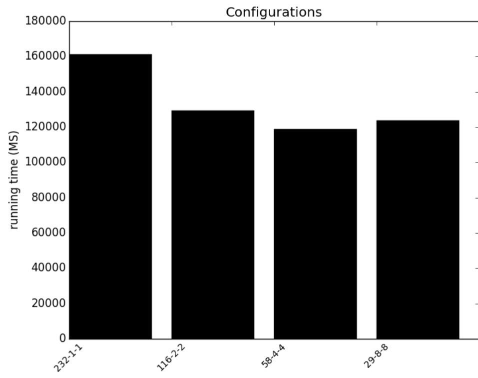
Splitting the same amount of resources

We study the four following configurations: 232-1-1, 116-2-2, 58-4-4 and 29-8-8. They all sum up to 232 GB RAM and 232 cores. those configurations gradually move from

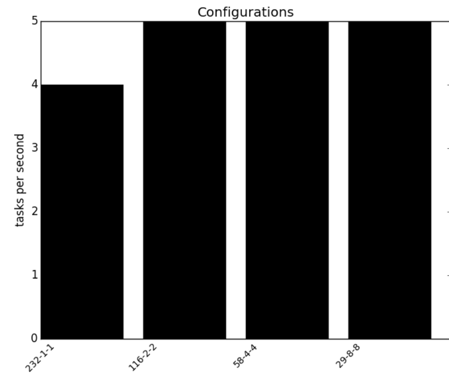
a large number of thin executors to a small number of big ones. We study the behaviour of Wordcount, Pagerank, and Connected Components.

Wordcount:

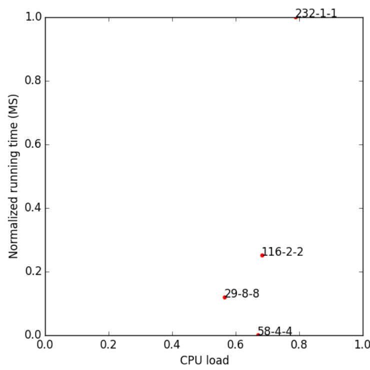
In figure 5.1, we observe that the delta in running time between 232-1-1 and 58-4-4 is about 25%. Looking at the double histogram 5.1c of CPU load to running time, we observe that 232-1-1 is also the one where executors are consuming the more CPU resource on average. 232-1-1 is behind in terms of throughput. All four configurations are memory efficient in the sense that they always use once close to their maximum heap allocation. Lastly, we note that the bigger the heap, the more time spent in garbage collection overall. We measure that time by adding up the time spent in GC across all executors.



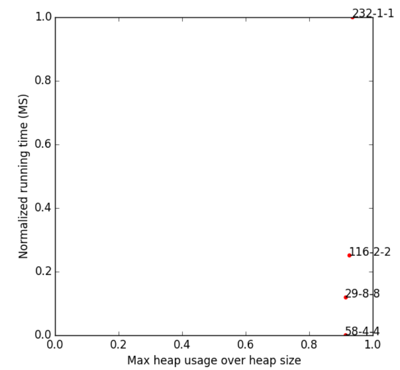
(a) Running time



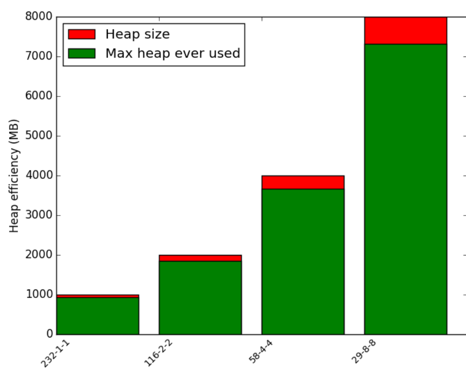
(b) Throughput



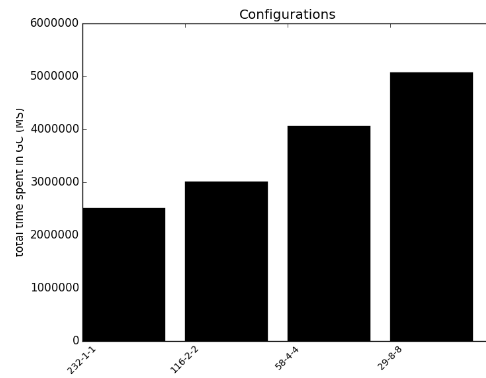
(c) CPU load to running time



(d) Heap usage to running time



(e) Max Heap to heap size

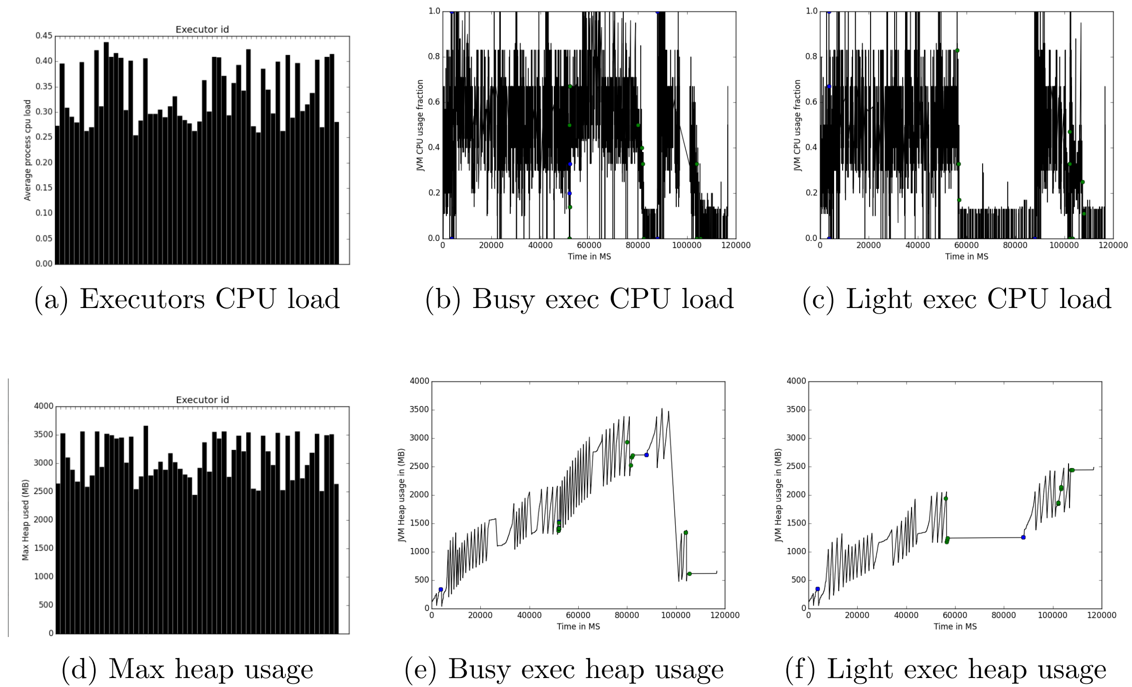


(f) Time spent in GC

FIGURE 5.1: Impact of executor configurations on Wordcount

As we are interested in finding out more precisely what makes 232-1-1 a worst

performer, and 58-4-4 a top one, we dive more closely into their respective monitoring data. figure 5.2 presents information such as the max heap per executor, as well as their average CPU load. Each bar of the bar plots represent an executor. We observe that there are some discrepancies in resource consumption. We find that there are some "light" and "busy" executors, and present some of their memory and CPU consumption life cycles. We observe that each time a new task is launched on the executor, alongside CPU activity, heap space is used as a new RDD partition is loaded in memory. Spark's LRU policy kicks in frequently, and we can see space freed as well. The reason for the difference between executors activity resides in the fact that, due to load imbalance, some executors are shipped a larger amount of tasks than others.

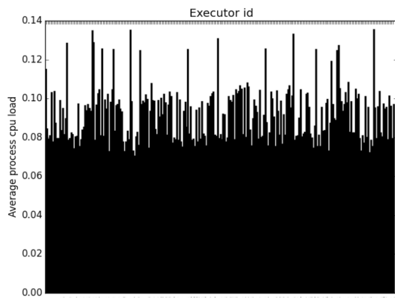


(a) Executors CPU load
(b) Busy exec CPU load
(c) Light exec CPU load
(d) Max heap usage
(e) Busy exec heap usage
(f) Light exec heap usage

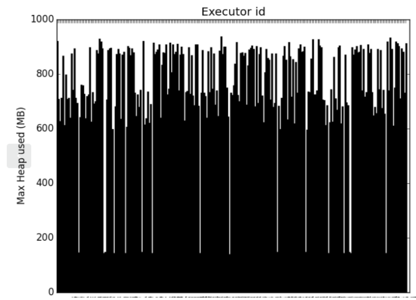
FIGURE 5.2: Wordcount details for 58-4-4. Blue dots represents tasks' start, green tasks' end

The scenario is resemblant for 232-1-1, but, in addition, there are also many

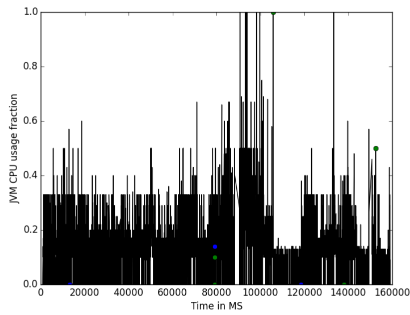
executors which were spawn, did nothing, and died. Figure 5.3 also presents graphs depicting such failed executors. The fact that they consume cpu time on the worker might explain why, in figure 5.1c, we observed that 232-1-1 had the higher CPU activity, in despite of its poor performances. We numbered 16 such executors.



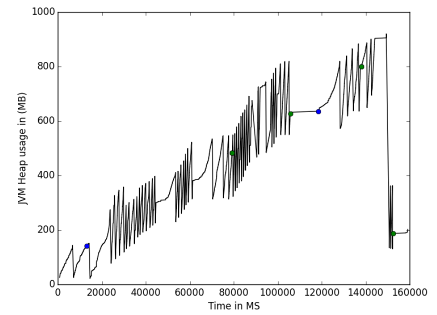
(a) CPU load



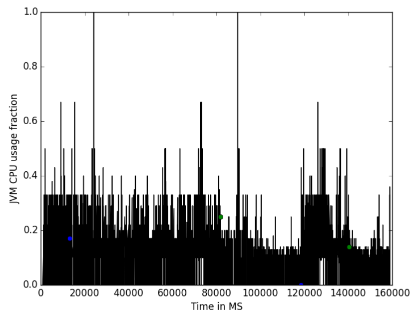
(b) Max heap usage



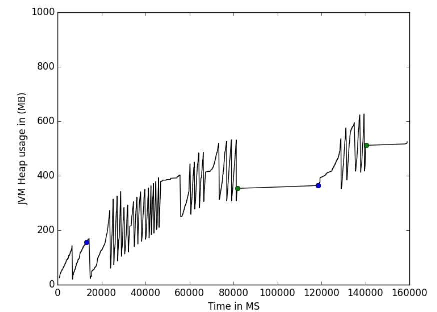
(c) Busy exec CPU load



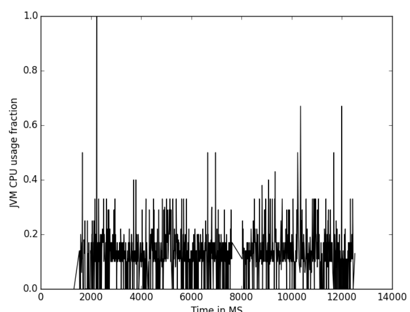
(d) Busy executor heap usage



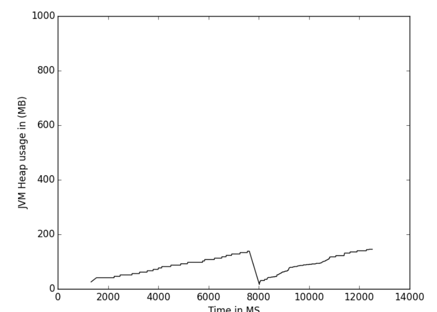
(e) Light exec CPU load



(f) Light exec heap usage



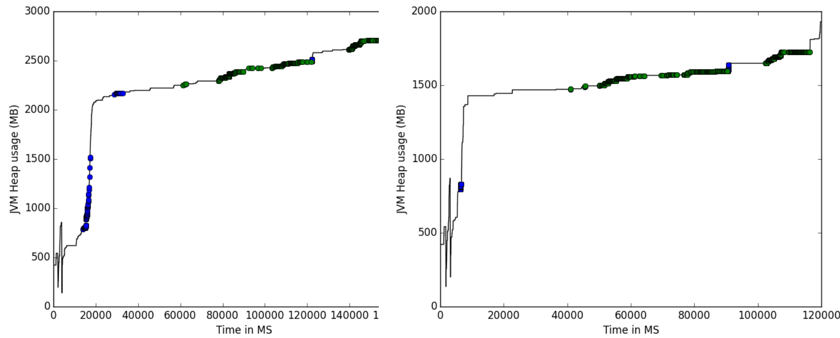
(g) Failed exec CPU load



(h) Failed exec heap usage

FIGURE 5.3: Wordcount details for 232-1-1. Blue dots represents tasks' start, green tasks' end

Lastly, we report, and present in figure 5.4, that the Driver is consuming more heap in the 232-1-1 case than it does with 58-4-4, which makes sense as it must coordinate much more independent actions that it has to with a smaller fleet of executors.



(a) 232-1-1 Driver heap usage (b) 58-4-4 Driver heap usage

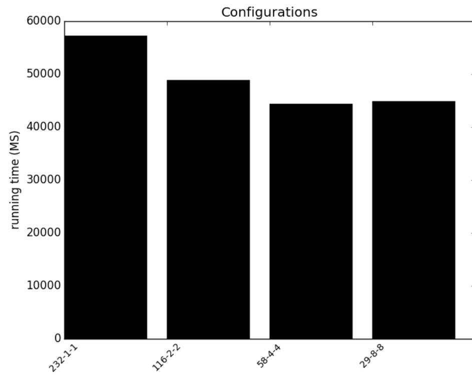
Figure 5.4: Drivers heap usage. Blue dots represents tasks' start, green tasks' end

Connected Components:

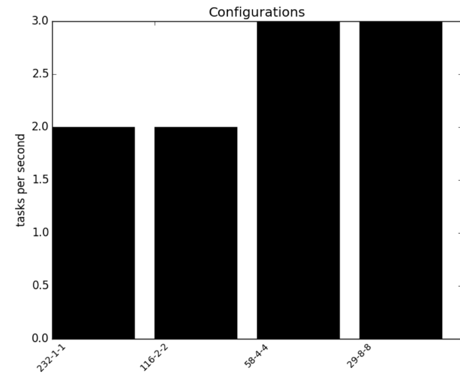
We now study the same configuration set with Connected Components. Figure 5.5 presents the same plot we shown for Wordcount. As we will detail soon, executors are, overall, very lightly loaded in this scenario. We observed that only a handful of executors is actually working for all configurations here. That means that in the case of 232-1-1, we have tons of idle executors, and observed that overall, more than 400 executors were spawned overall. As a result, we were not able to correctly compute the CPU load for it.

116-2-2, in the case of CC, has the same throughput than 232-1-1, and is also displaying the higher time spent in garbage collection overall. Surprisingly, 29-8-8 spent the smallest amount of time in GC. We can also see that the maximum heap ever used for all configurations is very low compared to their allocation. It seems

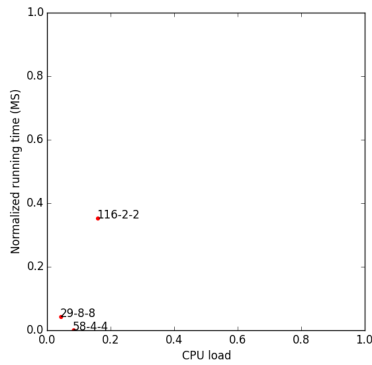
fair to assume that the dataset we performed Connected Component analysis on was small compared to the capacity of the cluster, and led to an overall important under utilization of its resources.



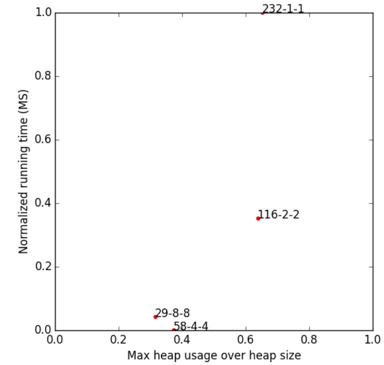
(a) Running time



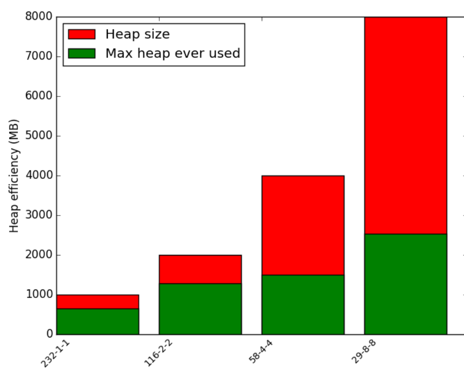
(b) Throughput



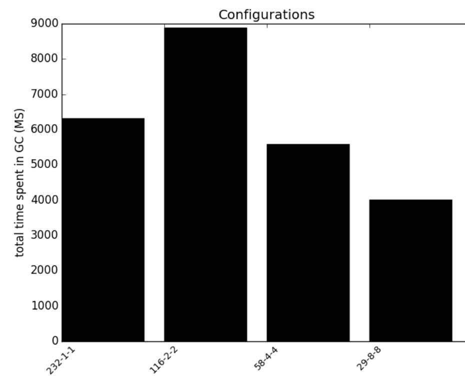
(c) CPU load to running time



(d) Heap usage to running time



(e) Max Heap to heap size

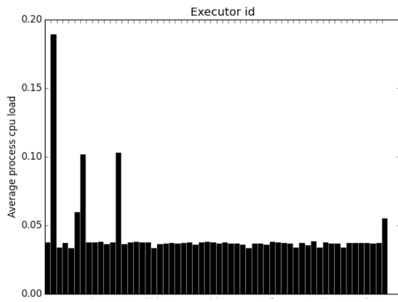


(f) Time spent in GC

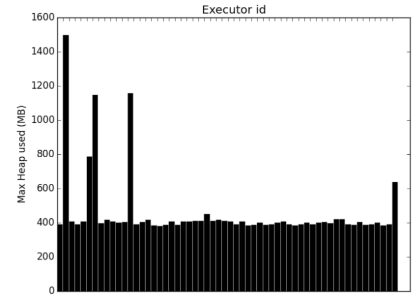
FIGURE 5.5: Impact of executor configurations on Connected Components

Figures 5.6 and 5.7 shows us the activity pattern for idle, light, and busy executors

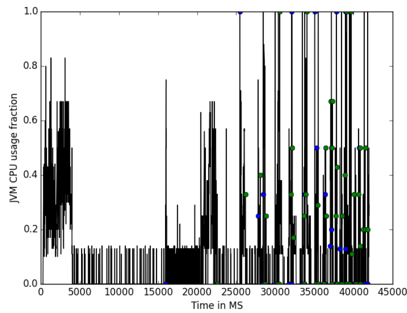
for 58-4-4 and 232-1-1, respectively. Even if their lifetime is very short, idle executors do consume CPU time. Taken at the scale of the hundreds of unecessary executors which were spawned, they do account for a lot of resource waste.



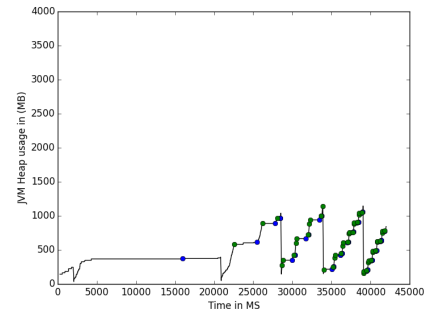
(a) CPU load



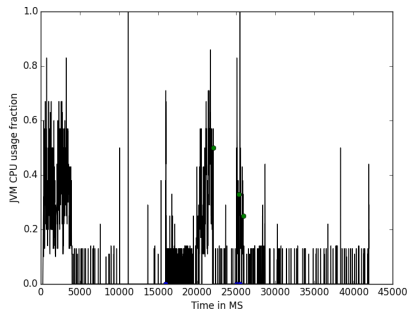
(b) Max heap usage



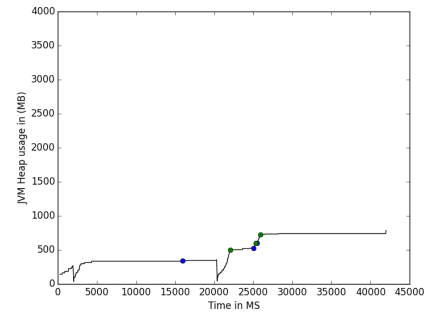
(c) Busy exec CPU load



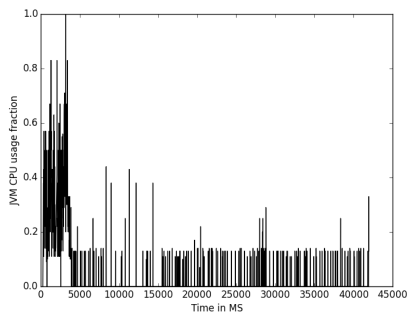
(d) Busy exec heap usage



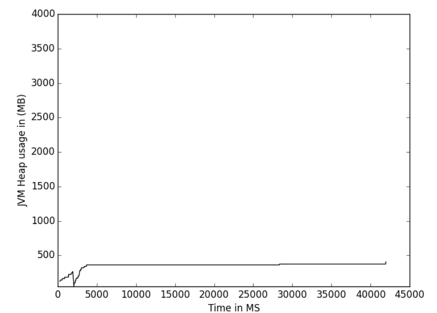
(e) Light exec CPU load



(f) Light exec heap usage



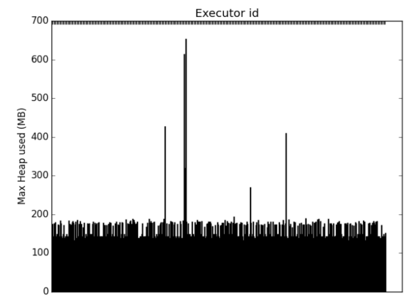
(g) Idle exec CPU load



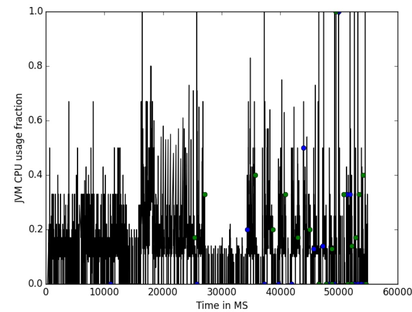
(h) Idle exec heap usage

Figure 5.6: CC details for 58-4-4. Blue dots represents tasks' start, green tasks' end

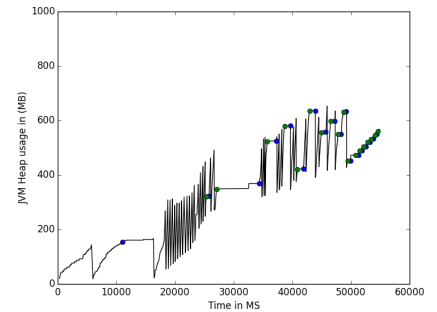
(a) CPU load



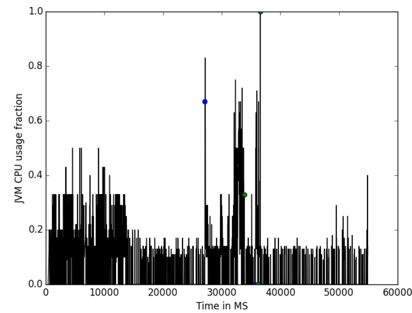
(b) Max heap usage



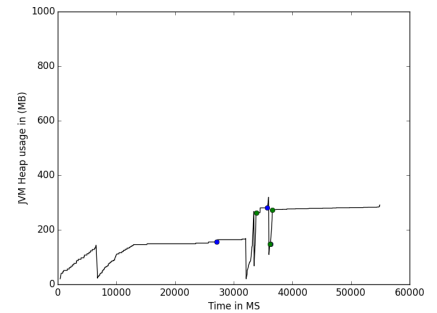
(c) Busy exec CPU load



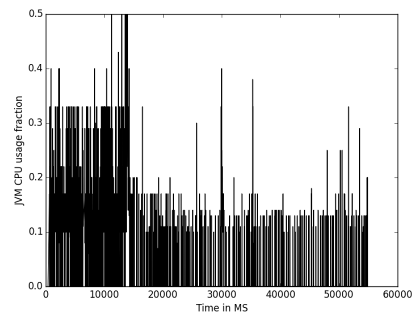
(d) Busy exec heap usage



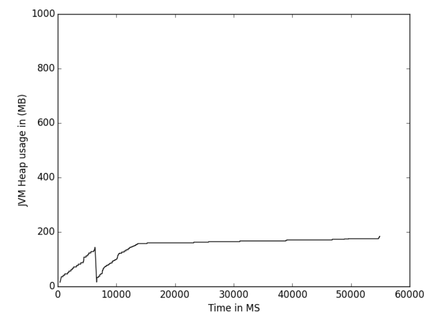
(e) Light exec CPU load



(f) Light exec heap usage



(g) Idle exec CPU load



(h) Idle exec heap usage

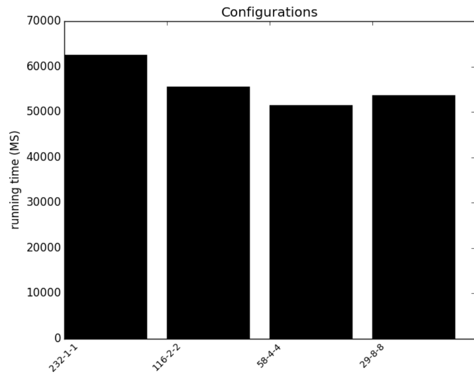
FIGURE 5.7: CC details for 232-1-1. Blue dots represents tasks' start, green tasks' end

Pagerank:

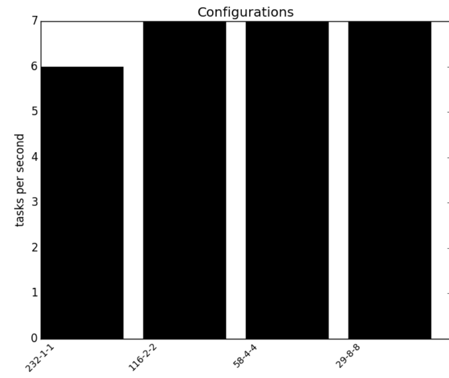
We conclude our study with Pagerank. We observe in figure 5.8 that the performers are pretty much in the same order. Memory utilization is very poor compared to the allocated amounts, which tends to indicate that Pagerank does not As a notable difference, 58-4-4 is also very high in average CPU load.

As for Connected Components, we observe a terrible imbalance in all the configurations. Figures 5.9 and 5.10 show that a few executors are outliers, in terms of resource consumption. Those Executors appear to be the only one, in each case, to do actual work after having received a task. We believe that the structure of the graph we use for the workloads is ... or delay scheduling...

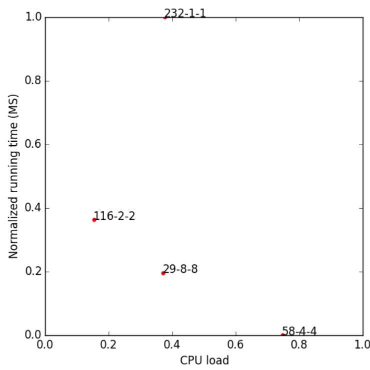
The reason why 58-4-4 display more discrepancy among its executor's cpu load is because many of them do not die shortly after creation, and keep consuming cpu time while not processing any task.



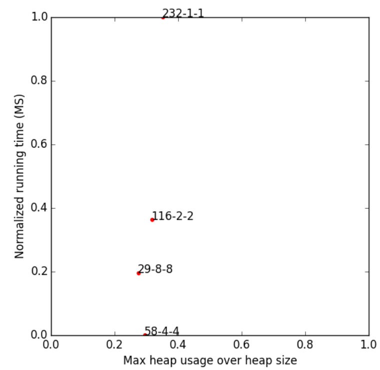
(a) Running time



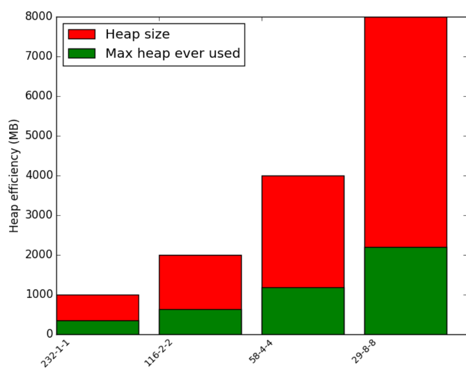
(b) Throughput



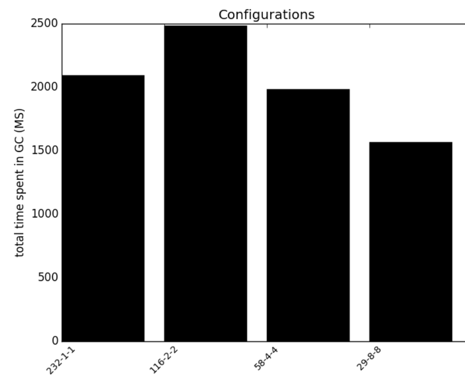
(c) CPU load to running time



(d) Heap usage to running time

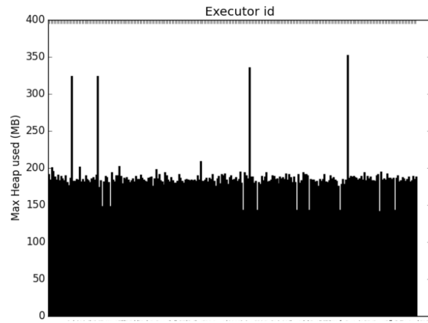


(e) Max Heap to heap size

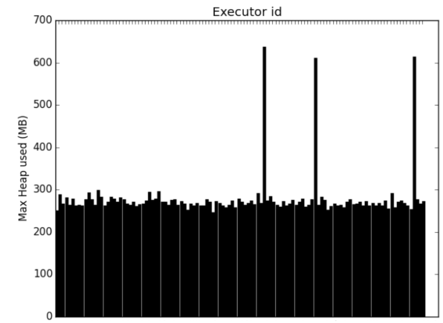


(f) Time spent in GC

FIGURE 5.8: Impact of executor configurations on Pagerank

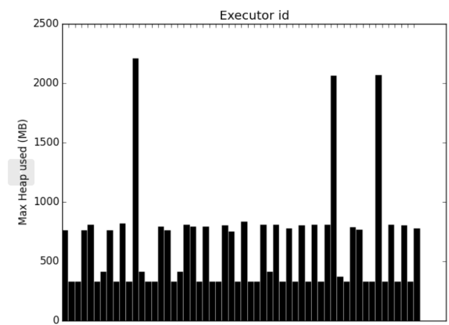


(a) 232-1-1 executors max heap



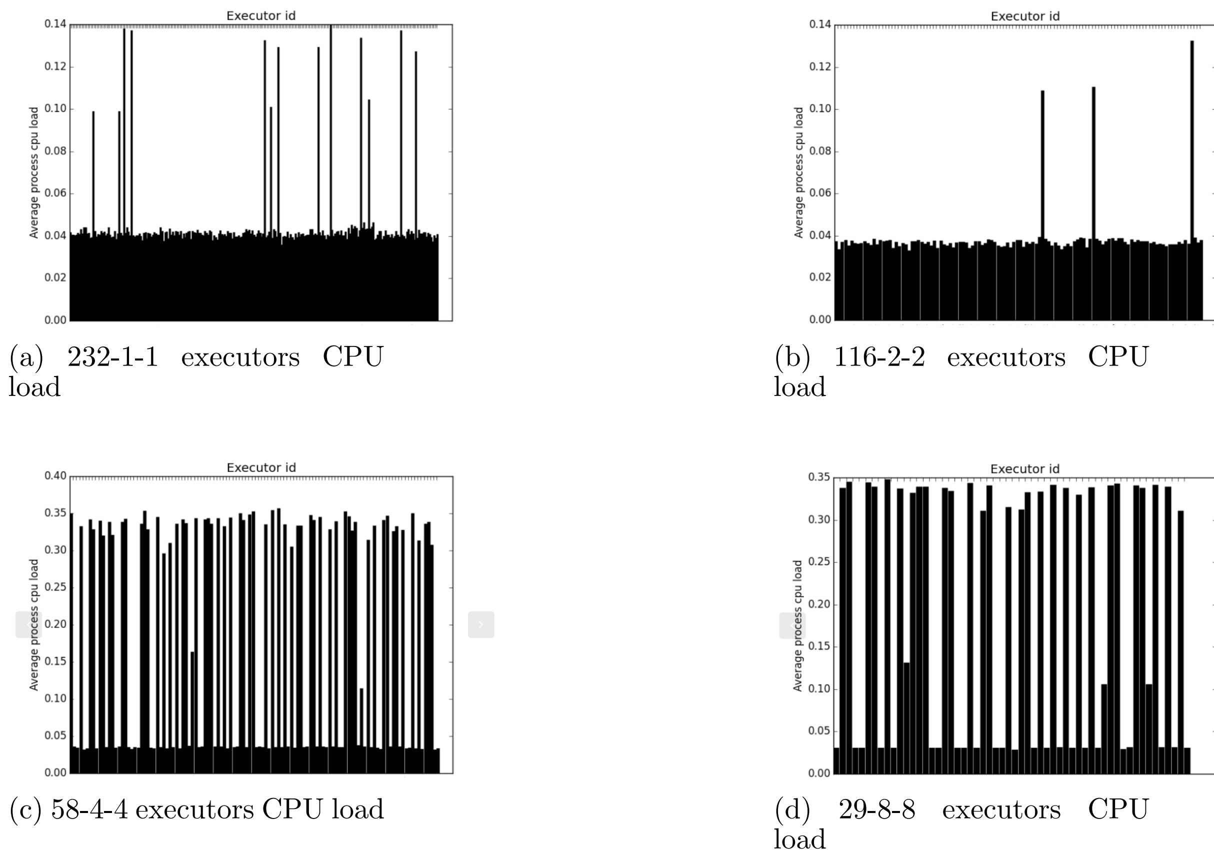
(b) 116-2-2 executors max heap

(c) 58-4-4 executors max heap



(d) 29-8-8 executors max heap

FIGURE 5.9: Maximum heap usage for executors across configurations for Pagerank



(a) 232-1-1 executors CPU load

(b) 116-2-2 executors CPU load

(c) 58-4-4 executors CPU load

(d) 29-8-8 executors CPU load

FIGURE 5.10: Average cpu load for executors across configurations for Pagerank

Comparing top and worst performers

Here we study differences between 2 top and 3 worst configurations observed for wordcount in our setup. We must be very careful when reading the plots, because the amount of resources per configuration, detailed in table 5.1, is not the same for each.

Figure 5.11 present them, by decreasing order of performance. The first thing we observe is that the running time of 29-14-6 and 57-7-3 are very similar, and that they have the exact same number of total resources, which might tone down the impact of partitioning on Wordcount performances, as long as executors have more than one thread. 116-1-2 (which is a worst performer), has significantly less memory

available than our two top performers, but more cores. 29-8-2 and 29-14-1, which have both more memory but less cores than 116-1-2, display an even slower running time. This suggests that Wordcount has a preferred ratio of memory to cores per executor, which we kind of hover in our experiments. In addition, we observe that it is the case that most the single threaded configurations, as 29-14-1, are worst performers.

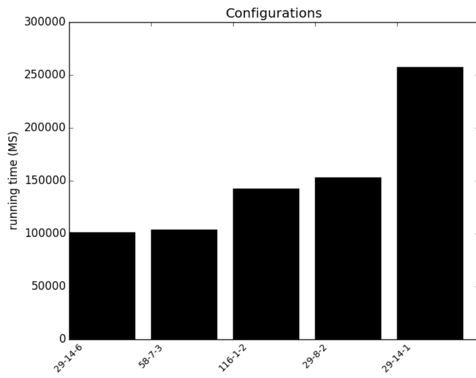
Looking at the CPU load and memory efficiency plots, we observe that there is a great difference between the two top performers, which is again not represented in running time. This might mean that the higher number of executor of 57-7-3 makes it up for their lower CPU usage. There is some kind of consistency between the worst performers CPU load and the fact that they run slowly. But 58-7-3 and 29-8-2 are not so far.

Discussion

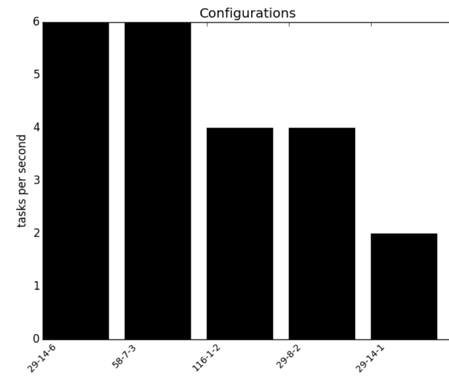
Overall, it seems that our workloads are under utilizing the cluster, for there are many idle executors. Also, those applications seem to run better on larger executors than on smaller ones, especially the graph analytic ones which concentrate work on very few executors.

We note that high cpu usage is not directly correlated with better performances, as when idle workers are spawned, they consume CPU time as well.

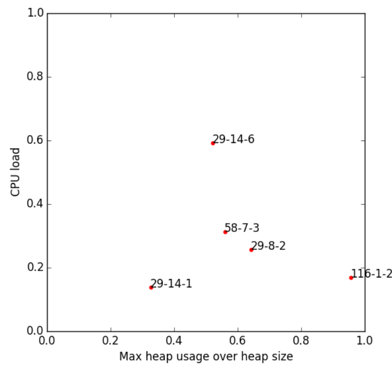
We also note that memory efficiency, measured by the ratio of maximum heap ever used by at least one executor to the heap allocated to any of them, does not seem to be representative of best or worst performances. We suspect that this metric is complicated to interpret because the maximum heap used at any time is dependent on garbage collection frequency, which is itself a function of heap size.



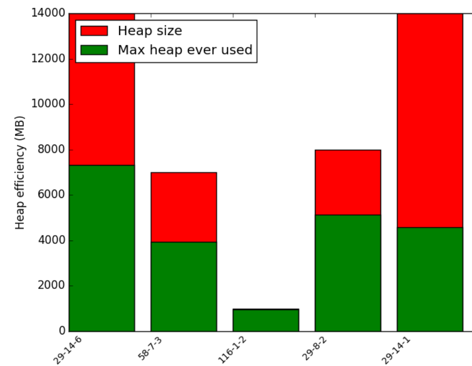
(a) Running time



(b) Throughput



(c) Memory efficiency and CPU load



(d) Heap usage

FIGURE 5.11: Top and worst configurations performance for Wordcount

Table 5.1: Configuration analyzed

Id	Configuration Set	Total resources
b	29-14-6	406 GB RAM, 174 cores
c	58-7-3	406 GB RAM, 174 cores
d	116-1-2	116 GB RAM, 232 cores
e	29-8-2	232 GB RAM, 58 cores
f	29-14-1	406 GB RAM, 29 cores

6

Conclusion

Future works will study a hybrid recommender where we combine IBCF and latent factor models to hopefully increase accury and tolerance to sparsity. We aim to implement the standard equation of Bellman et al. $\mu + b_i + b_u + q_i^T.p_u$, where μ is the baseline rating for the whole system, b_i the item bias, b_u the user bias, and $q_i^T.p_u$ the collaborative filtering part where we compute the dot product of the item's features vector with the user's one.

Recommender systems based on collaborative filtering make it difficult to make sense of what features and characteristics of the workloads are being manipulated, so we think such systems' scope should be narrowed to a subset of possible applications, which would have been previously filtered through some representative profiling methodology. Such methodology needs to be framework agnostic, but could be more strongly tied to the programming model in use. Another concern is the expansion of a configuration space which allows to transfer experience from a training set to an enlarged one, without losing confidence in the model.

Overall, repeating our experiments with many more applications and input size seems to be mandatory to refine further the choice of which model to pick.

7

Appendices

.1 Compute ranking score for a set of predictions

```
global_score = 0
for application in APPLICATIONS do
    application_score = 0
    for configuration in application.configurations() do
        configuration_score =configuration.predictedRank − 1
        for configuration_ranked_above in [1 .. configuration.predictedRank()]
do
            if        configuration_ranked_above.realRank()        <
configuration.realRank() then
                configuration_score − = 1
            end if
        end for
        application_score + = configuration_score
    end for
    application_score + = application_score / size(application.configurations())
end for
global_score = application_score / size(APPLICATIONS)
```

.2 Exhaustive description of our configuration spaces

Table 1: Appendix: Configuration spaces

Id	Configuration id	Total resources	Configuration id	Total resources
a	29-3-3	261 GB RAM, 87 cores		
	29-7-7	203 GB RAM, 203 cores		
	116-3-2, 58-6-4, 29-12-8	348 GB RAM, 232 cores		
	29-12-3, 87-4-1	348 GB RAM, 87 cores		
	116-3-1, 58-6-2, 29-12-4	348 GB RAM, 116 cores		
	29-5-5	145 GB RAM, 145 cores		
	29-9-6	261 GB RAM, 174 cores		
	232-1-1, 116-2-2, 58-4-4, 29-8-8	232 GB RAM, 232 cores		
	29-12-6, 58-6-3, 87-4-2	348 GB RAM, 174 cores		
	58-3-3, 29-6-6	174 GB RAM, 174 cores		
	29-14-7, 203-2-1	406 GB RAM, 203 cores		
	29-8-4	232 GB RAM, 116 cores		
	29-8-2	232 GB RAM, 58 cores		
	29-10-5	290 GB RAM, 145 cores		
b	58-7-2, 29-14-4	406 GB RAM, 116 cores	29-9-8	261 GB RAM, 232 cores
	29-8-4	232 GB RAM, 116 cores	29-11-3	319 GB RAM, 87 cores
	29-7-7	203 GB RAM, 203 cores	29-6-7	174 GB RAM, 203 cores
	29-9-7	261 GB RAM, 203 cores	29-1-5	29 GB RAM, 145 cores
	29-7-8	203 GB RAM, 232 cores	58-1-3, 29-2-6	58 GB RAM, 174 cores
	58-5-4, 29-10-8	290 GB RAM, 232 cores	29-14-3	406 GB RAM, 87 cores
	29-6-5	174 GB RAM, 145 cores	29-13-5	377 GB RAM, 145 cores
	29-4-7	116 GB RAM, 203 cores	29-1-8	29 GB RAM, 232 cores
	29-1-7	29 GB RAM, 203 cores	29-13-1	377 GB RAM, 29 cores
	29-10-3	290 GB RAM, 87 cores	29-11-5	319 GB RAM, 145 cores
	29-9-4	261 GB RAM, 116 cores	29-9-5	261 GB RAM, 145 cores
	29-8-1	232 GB RAM, 29 cores	29-14-5	406 GB RAM, 145 cores
	116-1-2, 58-2-4, 29-4-8	116 GB RAM, 232 cores	29-10-7	290 GB RAM, 203 cores
	203-2-1, 29-14-7	406 GB RAM, 203 cores	58-5-2, 29-10-4	290 GB RAM, 116 cores
	58-5-3, 29-10-6	290 GB RAM, 174 cores	29-11-2	319 GB RAM, 58 cores
	29-5-6	145 GB RAM, 174 cores	58-7-3, 29-14-6	406 GB RAM, 174 cores
	58-2-3, 29-4-6	116 GB RAM, 174 cores	29-9-6	261 GB RAM, 174 cores
	29-11-6	319 GB RAM, 174 cores	29-13-7	377 GB RAM, 203 cores
	29-11-7	319 GB RAM, 203 cores	29-13-6	377 GB RAM, 174 cores
	29-5-8	145 GB RAM, 232 cores	58-7-4, 29-14-8	406 GB RAM, 232 cores
	29-7-5	203 GB RAM, 145 cores	29-2-5	58 GB RAM, 145 cores
	29-11-8	319 GB RAM, 232 cores	29-1-6	29 GB RAM, 174 cores
	29-8-3	232 GB RAM, 87 cores	29-12-5	348 GB RAM, 145 cores
	29-3-7	87 GB RAM, 203 cores	29-12-1	348 GB RAM, 29 cores
	87-4-1, 29-12-3	348 GB RAM, 87 cores	29-5-5	145 GB RAM, 145 cores
	58-7-1, 29-14-2	406 GB RAM, 58 cores	58-5-1, 29-10-2	290 GB RAM, 58 cores
	29-3-8	87 GB RAM, 232 cores	58-4-3, 29-8-6	232 GB RAM, 174 cores
	29-8-2	232 GB RAM, 58 cores	29-11-1	319 GB RAM, 29 cores
	29-13-2	377 GB RAM, 58 cores	29-10-1	290 GB RAM, 29 cores
	58-1-4, 29-2-8	58 GB RAM, 232 cores	29-12-7	348 GB RAM, 203 cores
	29-3-6	87 GB RAM, 174 cores	29-3-5	87 GB RAM, 145 cores
	116-3-1, 58-6-2, 29-12-4	348 GB RAM, 116 cores	29-14-1	406 GB RAM, 29 cores
	29-13-4	377 GB RAM, 116 cores	29-8-7	232 GB RAM, 203 cores
	29-8-5	232 GB RAM, 145 cores	29-5-7	145 GB RAM, 203 cores
	58-6-1, 29-12-2	348 GB RAM, 58 cores	116-3-2, 58-6-4, 29-12-8	348 GB RAM, 232 cores
	29-9-1	261 GB RAM, 29 cores	29-13-8	377 GB RAM, 232 cores
	29-7-6	203 GB RAM, 174 cores	29-9-2	261 GB RAM, 58 cores
	58-3-3, 29-6-6	174 GB RAM, 174 cores	232-1-1, 116-2-2, 58-4-4, 29-8-8	232 GB RAM, 232 cores
	29-11-4	319 GB RAM, 116 cores	29-2-7	58 GB RAM, 203 cores
	29-9-3	261 GB RAM, 87 cores	87-4-2, 58-6-3, 29-12-6	348 GB RAM, 174 cores
	29-13-3	377 GB RAM, 87 cores	29-4-5	116 GB RAM, 145 cores
	29-10-5	290 GB RAM, 145 cores	58-3-4, 29-6-8	174 GB RAM, 232 cores

Bibliography

[1] http://hadoop.apache.org/docs/r2.6.0/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.

[2] http://bit.ly/1S33gH0.

[3] Apache flink, https://ci.apache.org/projects/flink/flink-docs-release-0.6/run_example_quickstart.html.

[4] http://aurora.apache.org/, Last visited in March 2016.

[5] http://codechannels.com/video/docker/docker/tupperware-containerized-deployment-at-facebook-by-aravind-narayanan/, Last visited in March 2016.

[6] http://snap.stanford.edu/data/web-google.html, Last visited in March 2016.

[7] https://www.docker.com/, Last visited in March 2016.

[8] https://zookeeper.apache.org/, Last visited in March 2016.

[9] H. J. Ahn. A new similarity measure for collaborative filtering to alleviate the new user cold-starting problem. *Information Sciences*, 178(1):37–51, 2008.

[10] A. W. M. Alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *Parallel and Distributed Systems, IEEE Transactions on*, 12(6):529–543, 2001.

[11] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 1–14. ACM, 2009.

[12] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, (12):33–37, 2007.

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

[14] C. Delimitrou and C. Kozyrakis. ibench: Quantifying interference for datacenter applications. In *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, pages 23–33. IEEE, 2013.

[15] C. Delimitrou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *ACM SIGARCH Computer Architecture News*, 41(1):77–88, 2013.

[16] D. G. Feitelson. Workload modeling for performance evaluation. In *Performance Evaluation of Complex Systems: Techniques and Tools*, pages 114–141. Springer, 2002.

[17] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[18] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, volume 11, pages 24–24, 2011.

[19] D. Goldberg, D. Nichols, B. M. Oki, and D. Terry. Using collaborative filtering

to weave an information tapestry. *Communications of the ACM*, 35(12):61–70, 1992.

[20] G. Guzun, J. E. Tosado, and G. Canahuate. Scalable preference queries for high-dimensional data using map-reduce. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 2243–2252. IEEE, 2015.

[21] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *CIDR*, volume 11, pages 261–272, 2011.

[22] M. Isard. Autopilot: automatic data center management. *ACM SIGOPS Operating Systems Review*, 41(2):60–67, 2007.

[23] N. S. Islam, M. Wasi-ur Rahman, X. Lu, D. Shankar, and D. K. Panda. Performance characterization and acceleration of in-memory file systems for hadoop and spark applications on hpc clusters. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 243–252. IEEE, 2015.

[24] Y. Koren. The bellkor solution to the netflix grand prize. *Netflix prize documentation*, 81, 2009.

[25] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[26] J. Mace, R. Roelke, and R. Fonseca. Pivot tracing: dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 378–393. ACM, 2015.

[27] J. Moore, J. Chase, K. Farkas, and P. Ranganathan. Data center workload monitoring, analysis, and emulation. In *Eighth Workshop on Computer Architecture Evaluation using Commercial Workloads*, 2005.

[28] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 293–307, 2015.

[29] S. Padala, D. Kumar, A. Raj, and J. Dharanipragada. Octopus: A multi-job scheduler for graphlab. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 293–298. IEEE, 2015.

[30] P. S. Rao and G. Porter. Is memory disaggregation feasible? a case study with spark sql. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2016.

[31] A. I. Schein, A. Popescul, L. H. Ungar, and D. M. Pennock. Methods and metrics for cold-start recommendations. In *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 253–260. ACM, 2002.

[32] T. Tang and G. McCalla. Utilizing artificial learners to help overcome the cold-start problem in a pedagogically-oriented paper recommendation system. In *Adaptive hypermedia and adaptive web-based systems*, pages 245–254. Springer, 2004.

[33] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache hadoop yarn: Yet another

resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

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

[35] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278. ACM, 2010.

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

[37] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 10:10, 2010.

[38] M. Zhang, J. Tang, X. Zhang, and X. Xue. Addressing cold start in recommender systems: A semi-supervised co-training algorithm. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, pages 73–82. ACM, 2014.