MULTI-CONSTRAINT SCHEDULING OF MAPREDUCE WORKLOADS

JORDÀ POLO

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor in Computer Science

Universitat Politècnica de Catalunya

2014

Jordà Polo: *Multi-constraint scheduling of MapReduce workloads,* A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

ADVISORS:
Yolanda Becerra
David Carrera

AFFILIATION:
Departament d'Arquitectura de Computadors
Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya

LOCATION:
Barcelona

ABSTRACT

In recent years there has been an extraordinary growth of large-scale data processing and related technologies in both, industry and academic communities. This trend is mostly driven by the need to explore the increasingly large amounts of information that global companies and communities are able to gather, and has lead the introduction of new tools and models, most of which are designed around the idea of handling huge amounts of data.

Alongside the need to manage ever larger amounts of information, other developments such as cloud computing have also contributed significantly to the growth of large-scale technologies. Cloud computing has dramatically transformed the way many critical services are delivered to customers, posing new challenges to data centers. As a result, there is a complete new generation of large-scale infrastructures, bringing an unprecedented level of workload and server consolidation, that demand not only new programming models, but also new management techniques and hardware platforms.

These infrastructures provide excellent opportunities to build data processing solutions that require vast computing resources. However, despite the availability of these infrastructures and new models that deal with massive amounts of data, there is still room for improvement, specially with regards to the integration of the two sides of these environments: the systems running on these infrastructures, and the applications executed on top of these systems.

A good example of this trend towards improved large-scale data processing is MapReduce, a programming model intended to ease the development of massively parallel applications, and which has been widely adopted to process large datasets thanks to its simplicity. While the MapReduce model was originally used primarily for batch data processing in large static clusters, nowadays it is mostly deployed along with other kinds of workloads in shared environments in which multiple users may be submitting concurrent jobs with completely different priorities and needs: from small, almost interactive, executions, to very long applications that take hours to complete. Scheduling and selecting tasks for execution is extremely relevant in MapReduce environments since it governs a job's opportunity to make progress and determines its performance. However, only basic primitives to prioritize between jobs are available at the moment, constantly causing either under or over-provisioning, as the amount of resources needed to complete a particular job are not obvious a priori.

This thesis aims to address both, the lack of management capabilities and the increased complexity of the environments in which MapReduce is executed. To that end, new models and techniques are introduced in order to improve the scheduling of MapReduce in the presence of different constraints found in real-world scenarios, such as completion time goals, data locality, hardware heterogeneity, or availability of resources. The focus is on improving the integration of MapReduce with the computing infrastructures in which it usually runs, allowing alternative techniques for dynamic management and provisioning of resources. More specifically, it is focused in three scenarios that are incremental in its scope. First, it studies the prospects of using high-level performance criteria to manage and drive the performance of MapReduce applications, taking advantage of the fact that MapReduce is executed in controlled environments in which the status of the cluster is known. Second, it examines the feasibility and benefits of making the MapReduce runtime more aware of the underlying hardware and the characteristics of applications. And finally, it also considers the interaction between MapReduce and other kinds of workloads, proposing new techniques to handle these increasingly complex environments.

Following these three items described above, this thesis contributes to the management of MapReduce workloads by 1) proposing a performance model for MapReduce workloads and a scheduling algorithm that leverages the proposed model and is able to adapt depending on the various needs of its users in the presence of completion time constraints; 2) proposing a new resource model for MapReduce and a placement algorithm aware of the underlying hardware as well as the characteristics of the applications, capable of improving cluster utilization while still being guided by job performance metrics; and 3) proposing a model for shared environments in which MapReduce is executed along with other kinds of workloads such as transactional applications, and a scheduler aware of these workloads and its expected demand of resources, capable of improving resource utilization across machines while observing completion time goals.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

1

INTRODUCTION

1.1 MOTIVATION

Current trends in computer science are driving users toward more service-oriented architectures such as the so-called cloud platforms, that allow provisioning of computing and storage, converting physical centralized resources into into virtual shared resources. The ideas behind it are not that different to previous efforts such as utility or grid computing, but thanks to the maturity of technologies like virtualization, cloud computing is becoming much more efficient: cost, maintenance and energy-wise.

At the same time, more business are becoming aware of the relevance of the data they are able to gather: from social websites to log files, there is a massive amount of information ready to be stored, processed, and exploited. Not so long ago it was relatively difficult to work with large amounts of data. The problem was not hard drive capacity, which has increased significantly over the years, but access speed, which improved at a much lower pace. However, new tools, most of which are originally designed and built around handling big amounts of data are making things easier. Developers are finally getting used to the idea of dealing with large datasets.

Both of these changes are not coincidental and respond to certain needs. On the one hand, nowadays it is much easier for companies to become global, target a larger number of clients and consequently deal with more data. On the other hand, the initial cost they are willing to afford keeps shrinking. Another issue that these new technologies help to address is that benefits may only arrive when dealing with sufficiently large data, but the upfront cost and the maintenance

of the large clusters required to process such datasets is usually prohibitive compared to the benefits.

Despite the availability of new tools and the shift to service-oriented computing, there is still room for improvement, specially with regards to the integration of the two sides of this kind of environment: the applications that provide services and the systems that run these applications.

Next-generation data centers will be composed of thousands of hybrid systems in an attempt to increase overall cluster performance and to minimize energy consumption. Applications specifically designed to make the most of very large infrastructures will be leveraged to develop massively distributed services. Thus, data centers will be able to bring an unprecedented degree of workload consolidation, hosting in the same infrastructure distributed services from many different users, and with completely different needs and requirements.

However, as of now, developers still need to think about the requirements of the applications in terms of resources (CPU, memory, etc.), and inevitably end up either under or over-provisioning. While nowadays it is relatively easy to update provisioning as needed in some service-oriented environments, for many applications this process is still manual and requires human intervention. Moving away from the old style way of managing resources is still a challenge. In a way, it can be though of as the equivalent of the revolution that the introduction of time-sharing supposed in the era of batch processing. Time-sharing allowed everyone to interact with computers as if they were the owners of the system. Likewise, freeing users from thinking about provisioning is the definite step to create the illusion of the cloud as an unlimited source of computing resources.

The main obstacle, though, is that the cloud is not actually an infinite and *free* source of computing resources: maintaining it is not trivial, resources are limited and providers need some way to prioritize services. If users are freed of the task of provisioning, then there must be some other mechanism to make both, sharing and accounting, possible. On the other hand, some parts of these systems seem to be ready for this shift, specially the lower level components and the middleware. But the applications that run the services on top of cloud platforms seem to be lagging behind. As a relatively young development platform, it is to be expected that not all applications are fully integrated, but it seems clear that these represent the next and most obvious target in order to consolidate this kind of platforms.

One example of the kind of application that is worth targeting is the MapReduce programming framework. The MapReduce model allows developers to write massively parallel applications without much effort, and is becoming an essential tool in the software stack of many companies that need to deal with large datasets. MapReduce fits well

with the idea of dynamic provisioning, as it may run on a large number of machines and is already widely used in cloud environments. Additionally, frameworks like MapReduce also represent the kind of component that could benefit the most from a better integration with the computing environment since it is not only the framework itself that takes advantage of it: it is beneficial for all the applications that are based on the framework.

MapReduce is still mostly deployed as a tool for batch processing in large, static clusters. The same physical resources may be shared between a number of users with different needs, but only basic primitives to prioritize between them are available. This constantly causes either under- or over-provisioning, as the amount of resources needed to do a particular job are not obvious a priori.

This thesis aims to study and address these problems with the idea of improving the integration of MapReduce with the computing platforms in which it is executed, allowing alternative techniques for dynamic management and provisioning of resources. More specifically, it is focused in three areas. First, it will explore the prospects of using high-level performance criteria to manage and drive the performance of MapReduce applications, taking advantage of the fact that MapReduce is executed in controlled environments in which the status of the cluster is known. Second, it will explore the feasibility and benefits of making the MapReduce runtime more aware of the underlying hardware and the characteristics of applications. And finally, it will study the interaction between MapReduce and other kinds of workloads, proposing new techniques to handle these increasingly complex environments.

1.2 CONTRIBUTIONS

The contributions of this thesis revolve around the same guiding theme: management and scheduling of environments in which MapReduce plays a central role. All the work done is incremental in that each contribution is based on the previous one, but at the same time each one of them delves into a new topic and proposes solutions to different problems.

Figure 1.1 illustrates the three main directions this thesis explores, and the steps taken in each one of them. The first direction represents scheduling MapReduce environments with Time Constraints, the second direction represents scheduling with Space Constraints and the third direction scheduling in Shared Environments. While each direction represents a contribution, the strong point of this thesis is combination of all of them. More details about each contribution are provided in the following sections.

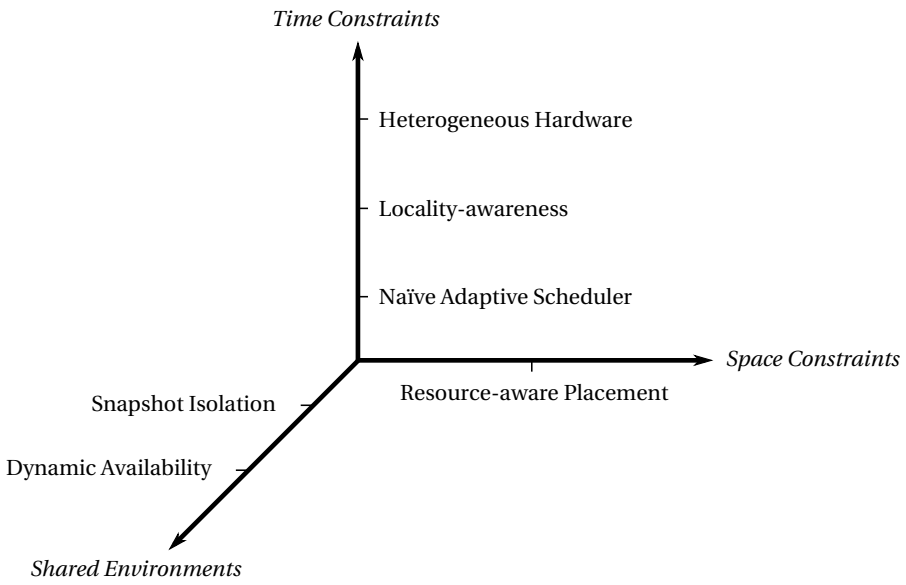


Figure 1.1: Major steps for each contribution

1.2.1 *Scheduling with Time Constraints*

While the MapReduce model was originally used primarily for batch data processing in large, static clusters, nowadays it is mostly used in shared environments in which multiple users may submit concurrent jobs with completely different priorities and needs: from small, almost interactive, executions, to very long programs that take hours to complete. Scheduling and selecting tasks for execution is extremely relevant in MapReduce environments since it governs a job's opportunity to make progress and influence its performance. However, only basic primitives to prioritize between them are available, constantly causing either under or over-provisioning, as the amount of resources needed to complete a particular job are not obvious a priori.

In these highly dynamic, shared, multi-user and multi-job environments new jobs can be submitted at any time, and the actual amount of resources available for each application can vary over time. Defining time constraints, in the form of user-specified job completion time goals, is necessary to predict and manage the performance of these kinds of workloads efficiently.

The **first contribution** of this thesis is a performance model for MapReduce workloads and a scheduling algorithm that leverages the proposed model and allows management of workloads with time constraints. The proposed scheduler is driven by continuous job performance management and is able to adapt depending on the needs of the users. It dynamically predicts the performance of concurrently running MapReduce jobs and adjusts its allocation, allowing appli-

cations to meet their completion time goals. The scheduler relies on estimates of individual job completion times given a particular resource allocation, and uses these estimates so as to maximize each job's chances of meeting its goals.

The scheduler is also capable of taking into account other requirements, such as data locality and hardware affinity. The former ensures that jobs are run as close as possible to where the data is located in the cluster, as long as it does not have a negative impact on the performance. The latter allows running MapReduce on heterogeneous clusters, making it possible to provide performance estimates depending on the machines in which jobs being executed.

As shown by experiments run on real clusters, this reactive approach in which the scheduler dynamically and automatically adapts to different kinds of jobs is one of the keys to successfully schedule workloads in this kind of ever-changing environments. It has also proven to be an improvement over previous MapReduce scheduling architectures, allowing a much more intuitive and accurate way to prioritize jobs.

The work performed in this area has resulted in the following main publications:

[62] Jordà Polo, David Carrera, Yolanda Becerra, Jordi Torres, Eduard Ayguadé, Malgorzata Steinder, and Ian Whalley. Performance-driven task co-scheduling for MapReduce environments. In *Network Operations and Management Symposium, NOMS*, pages 373–380, Osaka, Japan, 2010

[61] Jordà Polo, David Carrera, Yolanda Becerra, Jordi Torres, and Eduard Ayguadé. Performance Management of Accelerated MapReduce Workloads in Heterogeneous Clusters. In *ICPP '10: Proceedings of the 39th IEEE/IFIP International Conference on Parallel Processing*, San Diego, CA, USA, 2010

[64] Jordà Polo, Yolanda Becerra, David Carrera, Malgorzata Steinder, Ian Whalley, Jordi Torres, and Eduard Ayguadé. Deadline-Based MapReduce Workload Management. *IEEE Transactions on Network and Service Management*, pages 1–14, 2013-01-08 2013. ISSN 1932-4537

1.2.2 *Scheduling with Space and Time Constraints*

MapReduce is widely used in shared environments in which applications share the same physical resources, in line with recent trends in data center management which aim to consolidate workloads in order to achieve cost and energy savings. At the same time, next generation data centers now include heterogeneous hardware in order to run specialized workloads more efficiently. This combination of heterogeneous workloads and heterogeneous hardware is challenging because MapReduce schedulers are not aware of the resource capacity of the cluster nor the resource needs of the applications. Resource

management is therefore increasingly important in this scenario since users require both high levels of automation and resource utilization while avoiding bottlenecks.

The **second contribution** of this thesis is a new resource model for MapReduce and a scheduler based on a resource-aware placement algorithm that leverages the proposed model. The scheduler is aware of the underlying hardware as well as the characteristics of the applications, and is capable of improving cluster utilization while still being guided by job performance metrics. The scheduler builds upon the one presented in the first contribution, which is also guided by completion goals, but we take this idea a step further in order to allow space constraints in addition to time constraints. These space constraints, given by the availability of resources and the characteristics of MapReduce jobs, are intended to help the scheduler make more efficient placement decisions automatically.

In order to achieve both principles, resource awareness and continuous job performance management, a new resource model for MapReduce frameworks is introduced. Unlike the basic MapReduce model, this proposal provides a more fine-grained approach that leverages resource profiling information to obtain better utilization of resources and improve application performance. At the same time, it adapts to changes in resource demand by allocating resources to jobs dynamically.

This contribution represents a novel model of MapReduce scheduling since it makes it possible to decide not only how many resources are allocated to reach certain time constraints, but also how and where in the cluster tasks should be running in order to maximize resource utilization. This more proactive approach allows for the formulation of a placement problem solved by means of a utility-driven algorithm, which in turn provides the scheduler with the adaptability needed to respond to changing conditions in resource demand and availability. To measure and compare the performance of jobs, the scheduler uses a utility function that combines the resources allocated to that job with its completion time goal and job characteristics.

The work performed in this area has resulted in the following publication:

[63] Jordà Polo, Claris Castillo, David Carrera, Yolanda Becerra, Ian Whalley, Malgorzata Steinder, Jordi Torres, and Eduard Ayguadé. Resource-Aware Adaptive Scheduling for MapReduce Clusters. In *ACM IFIP USENIX 12th International Middleware Conference*, pages 187–207, Lisbon, Portugal, 2011. Springer. ISBN 978-3-642-25820-6. doi: 10.1007/978-3-642-25821-3_10

1.2.3 *Scheduling with Space and Time Constraints in Shared Environments*

The last part of this thesis focuses on a scenario that is becoming increasingly important in data centers. Instead of running on dedicated machines, MapReduce is executed along with other resource-consuming workloads, such as transactional applications. All workloads may potentially share the same data store, some of them consuming data for analytics while others may be acting as data generators. Twitter, Facebook, and other companies that need to handle large amounts of data, accessing and processing it in different ways and for different purposes, follow this approach of sharing multiple workloads on the same data center.

These shared environments involve higher workload consolidation, which helps improve resource utilization, but is also challenging due to the interaction between workloads of very different nature. One of the major issues found in this scenario is related to the integration of the storage. Storage is a key component since it usually deals with multiple producers and consumers of data, and often serves different kinds of workloads at the same time: from responding to transactional queries to storing the output of long-running data analytics jobs, each one of them with slightly different needs.

There are also other issues that arise when multiple workloads are collocated sharing the same machines. MapReduce schedulers, for instance, assume that the amount of available resources remains the same over time, but resources are no longer stable in a shared environment with transactional workloads, which are known to be bursty and have a varying demand over time. Hence, this scenario requires deep coordination between management components, and single applications can not be considered in isolation but in the full context of mixed workloads in which they are deployed.

The **third contribution** of this thesis is twofold: first, a scheduler and performance model for shared environments, and second, the necessary snapshotting mechanisms to allow the shared data store to be used by both transactional and analytics workloads.

The proposed scheduler aims to improve resource utilization across machines while observing completion time goals, taking into account the resource demands of non-MapReduce workloads, and assuming that the amount of resources made available to the MapReduce applications is dynamic and variable over time. This is achieved thanks to a new algorithm that provides a more proactive approach for the scheduler to estimate the need of resources that should be allocated to each job.

The work performed in this area has resulted in the following main publications:

[65] Jordà Polo, David Carrera, Yolanda Becerra, Malgorzata Steinder, Mike Spreitzer, Jordi Torres, and Eduard Ayguadé. Enabling Distributed Key-Value Stores with Low Latency-Impact Snapshot Support. In *Proceedings of the 12th IEEE International Symposium on Network Computing and Applications (NCA 2013)*, Boston, MA, USA, 2013. IEEE Computer Society

[66] Jordà Polo, David Carrera, Yolanda Becerra, Jordi Torres, Eduard Ayguadé, and Malgorzata Steinder. Adaptive MapReduce Scheduling in Shared Environments. In *Proceedings of the 14th IEEE ACM International Symposium On Cluster, Cloud And Grid Computing (CCGrid 2014)*, Chicago, IL, USA, 2014. IEEE Computer Society

1.3 THESIS ORGANIZATION

The remaining chapters of this thesis are organized as follows. Chapter 2 introduces some basic concepts related to MapReduce, datastorages, and hardware heterogeneity. Chapter 3 a introduces a scheduler for multi-job MapReduce workloads that is able to dynamically build performance models of applications, and then use them for scheduling purposes. Chapter 4 presents a resource-aware scheduling technique for MapReduce multi-job workloads that aims at improving resource utilization across machines while observing completion time goals. Chapter 5 is focused in how to improve the management of MapReduce in the shared environments in which it is usually executed. And finally, Chapter 6 presents the conclusions and future work of this thesis.

1.4 ACKNOWLEDGEMENTS

This work is partially supported by the Ministry of Science and Technology of Spain and the European Union's FEDER funds (TIN2007-60625, TIN2012-34557), by the Generalitat de Catalunya (2009-SGR-980), by the BSC-CNS Severo Ochoa program (SEV-2011-00067), by the by the European Commission's IST activity of the 7th Framework Program under contract number 317862, by IBM through the 2008 and 2010 IBM Faculty Award programs, and by the HiPEAC European Network of Excellence (IST-004408).

2

BACKGROUND

2.1 PROCESSING DATA WITH MAPREDUCE

MapReduce [24] is a programming model used to develop massively parallel applications that process and generate large amounts of data. It was first introduced by Google in 2004, and has since become an important tool for distributed computing. It is specially suited to operate on large datasets on clusters of computers, as it is designed to tolerate machine failures.

Essentially, MapReduce divides the work into small computations in two major steps, *map* and *reduce*, which are inspired by similar primitives that can be found in LISP and other functional programming languages. The input is formed by a set of key-value pairs, which are processed using the user-defined map function to generate a second set of intermediate key-value pairs. Intermediate results are then processed by the reduce function, which merges values by key.

While MapReduce is not something entirely new nor a revolutionary concept, it has helped to standardize parallel applications. And even though its interface is simple, it has proved to be powerful enough to solve a wide-range of real-world problems: from web indexing to image analysis to clustering algorithms.

MapReduce provides high scalability and reliability thanks to the division of the work into smaller units. Jobs are submitted to a master node, which is in charge of managing the execution of applications in the cluster. After submitting a job, the master initializes the desired number of smaller tasks or units of work, and puts them to run on worker nodes. First, during the map phase, nodes read and apply the map function to a subset of the input data. The map's partial output

is stored locally on each node, and served to worker nodes executing the reduce function.

Input and output files are usually stored in a distributed file system, but in order to ensure scalability, the master tries to assign *local* work, meaning the input data is available locally. On the other hand, if a worker node fails to deliver the unit of work it has been assigned to complete, the master node is always able to send the work to some other node.

2.1.1 *A Sample Application*

MapReduce is currently being used for many different kinds of applications, from very simple helper tools that are part of a larger environment, to more complete and complex programs that may involve multiple, chained MapReduce executions. This section includes a description of a typical MapReduce application, a word count, following the steps from the input to the final result.

The goal of a word count application is to get the frequency of words in a collection of documents. Word count was the problem that exemplified MapReduce in the original paper [24], and has since become the canonical example to introduce how MapReduce works.

To compute the frequency of words, a sequential program would need to read all the documents, keeping a list of <word, count> pairs, incrementing the appropriate count value every time a word is found. As you will see below, MapReduce's approach is slightly different. First of all, the problem is divided into two stages known as *map* and *reduce*, named after the functions that are applied while they are in progress. The `map()` function is applied to every single element of the input, and since there is no need to do so in any particular order, it effectively makes it possible to parallelize all the work. For each element, `map()` emits key-value pairs to be worked on later during the reduce stage. The generated key-value pairs are grouped and processed by key, so for every key there will be a list of values. The `reduce()` function is applied to these lists of values produced during the map stage, and provides the final result.

Listings 1 and 2 show how these functions are implemented in an application such as word count. The `map()` is simple: it takes a line of the input, splits it into words, and for each word emits a <word, count> key-value pair, where count is the partial count and thus always 1. Note that in this example the input is split into lines, but it could have been split into some other identifiable unit (e.g. paragraphs, documents, etc).

Listing 1: Word count: `map()` function

```
// i: key -- can be ignored in this example
// line: line contents
```

```
void map(string i, string line):
        for word in line:
                print word, 1
```

The reduce function takes <key, list(values)> pairs and goes through all the values to get the aggregated result for that particular key.

Listing 2: Word count: `reduce()` function

```
// word: the key
// partial_counts: a list of partial count values
void reduce(string word, list partial_counts):
        total = 0
        for c in partial_counts:
                total += c
        print word, total
```

A good exercise to understand how data is processed by MapReduce is to follow step by step how a small input evolves into the final output. For instance, imagine that the input of the word count program is as follows:

Listing 3: Word count: sample input

```
Hello World
Hello MapReduce
```

Since in this example the `map()` function is applied to every line, and the input has two lines, it is possible to run two `map()` functions simultaneously. Each function will produce a different output, but the format will be similar: <word, 1> pairs for each word. For instance, the `map()` reading the first line will emit the following partial output:

Listing 4: Word count: intermediate output of the first map

```
Hello, 1
World, 1
```

Finally, during the reduce stage, the intermediate output is *merged* grouping outputs by key. This results in new pairs formed by key and lists of values: <Hello, (1, 1)>, <World, (1)>, and <MapReduce, (1)>. These pairs are then processed by the `reduce()` function, which aggregates the lists and produces the final output:

Listing 5: Word count: final output

```
Hello, 2
World, 1
MapReduce, 1
```

Word count is an interesting example because it is simple, and the logic behind the `map()` and `reduce()` functions is easy to understand. As can be seen in the following examples, MapReduce is able to compute a lot more than a simple word count, but even though it is possible to make these functions more complex, it is recommended to keep them as simple as possible to help distribute the computation. If need be, it is always possible to chain multiple executions, using the output of one application as the input of the next one.

On the other hand, MapReduce may seem a bit overkill for a problem like word counting. For one thing, it generates huge amounts of intermediate key-value pairs, so it may not be entirely efficient for small inputs. But it is designed with scalability in mind, so it begins to make sense as soon as the input is large enough. Besides, most MapReduce programs also require some level of tweaking on both the application itself and on the server side (block size, memory, etc). Some of these refinements are not always obvious, and it is usually after a few iterations that applications are ready to be run on production.

It should also be noted this example is focused on the MapReduce computation, and some steps such as input distribution, splitting, and reduce partitioning are intentionally omitted, but will be described later.

2.1.2 *Examples of Use*

MapReduce is specially well suited to solve *embarrassingly* parallel problems, that is, problems with no dependencies or communication requirements in which it is easy to achieve a speedup proportional to the size of the problem when it is parallelized.

Below is the description of some of the main problems (not necessarily embarrassingly parallel) and areas where MapReduce is currently used.

2.1.2.1 *Distributed Search and Sort*

Besides the beforementioned word frequency counting application, searching and sorting are some of the most commonly used examples to describe the MapReduce model. All these problems also share the fact that they are helper tools, thought to be integrated into larger environments with other applications, very much like their pipeline-based UNIX-like equivalent tools: `wc`, `grep`, `sort`, etc. Moreover, knowing how these problems are implemented in MapReduce can be of great help to understand it, as they use different techniques.

A distributed version of `grep` is specially straightforward to implement using MapReduce. Reading line by line, maps only emit the current line if it matches the given pattern. And since the map's in-

termediate output can be used as the final output, there is no need to implement the `reduce()` function.

Sorting is different from searching in that the map stage only reads the input and emits everything (identity map). If there is more than one reducer and the output is supposed to be sorted globally, the important part is how to get the appropriate key and partition the input so that all keys for a particular reducer N come before all the keys for the next reducer $N + 1$. This way the output of the reducers can be numbered and concatenated after they are all finished.

2.1.2.2 *Inverted Indexing and Search Engines*

When Google's original MapReduce implementation was completed, it was used to regenerate the index of their search engine. Keeping indexes up to date is one of the top priorities of Internet search engines, but web pages are created and updated every day, so an scalable solution is a must.

Inverted indexes are one of the typical data structures used for information retrieval. Basically, an inverted index contains a list of references to documents for each word. To implement an inverted index with MapReduce, the map reads the input and for each words emits the document ID. Reduces then read it and output words along with the list of documents in which they appear.

Other than Google, other major search engines such as Yahoo! are also based on MapReduce. The need to improve the scalability of the Free, open-source software search engine Nutch also promoted the foundation of Hadoop, one of the most widely used MapReduce implementations to date.

2.1.2.3 *Log Analysis*

Nowadays service providers generate large amounts of logs from all kinds of services, and the benefits of analyzing them are to be found when processing them *en masse*. For instance, if a provider is interested in tracking the behaviour of a client during long periods of time, reconstructing user sessions, it is much more convenient to operate over all the logs.

Logs are a perfect fit for MapReduce for other reasons too. First, logs usually follow a certain pattern, but they are not entirely structured, so it is not trivial to use a RDBMS to handle them and may require changes to the structure of the database to compute something new. Secondly, logs represent a use case where scalability not only matters but is a key to keep the system sustainable. As services grow, so does the amount of logs and the need of getting something out of them.

Companies such as Facebook and Rackspace use MapReduce to examine log files on a daily basis and generate statistics and on-demand analysis.

2.1.2.4 *Graph Problems*

MapReduce is not perfectly fit for all graph problems, as some of them require walking through the vertices, which will not be possible if the mappers receive only a part of the graph, and it is not practical to receive the whole graph as it would be way too big to handle and require a lot of bandwidth to transfer. But there are ways to work around these issues [22] such as using multiple map and reduce iterations, along with custom optimized graph representations such as sparse adjacency matrices.

A good example of an Internet-scale graph problem solved using MapReduce is PageRank, an algorithm that ranks interlinked elements. PageRank can be implemented as a chained MapReduce application that at each step iterates over all the elements calculating its PageRank value until converging.

2.1.3 *Comparison with Other Systems*

Analyzing and performing computations on massive datasets is not something new, but it is not easy to compare MapReduce to other systems since it is often used to do things in a way that simply was not possible before using standardized tools. But besides creating a new *market*, MapReduce is also drawing the attention of developers, who use it for a wide range of purposes. The following comparison describes some of the technologies that share some kind of the functionality with MapReduce.

2.1.3.1 *RDBMS*

Relational Database Management Systems are the dominant choice for transactional and analytical applications, and they have traditionally been a well-balanced and good enough solution for most applications. Yet its design has some limitations that make it difficult to keep the compatibility and provide optimized solution when some aspects such as scalability are the top priority.

There is only a partial overlap of functionality between RDBMSs and MapReduce: relational databases are suited to do things for which MapReduce will never be the optimal solution, and vice versa. For instance, MapReduce tends to involve processing most of the dataset, or at least a large part of it, while RDBMS queries may be more fine-grained. On the other hand, MapReduce works fine with semi-structured data since it is interpreted while it is being processed, unlike RDBMSs, where well structured and normalized data is the

key to ensure integrity and improve performance. Finally, traditional RDBMSs are more suitable for interactive access, but MapReduce is able to scale linearly and handle larger datasets. If the data is large enough, doubling the size of the cluster will also make running jobs twice as fast, something that is not necessarily true of relational databases.

Another factor that is also driving the move toward other kind of storage solutions are disks. Improvements in hard drives seem to be relegated to capacity and transfer rate only. But data access in a RDBMS is usually dominated by seek times, which have not changed significantly for some years. Solid-state drives may prove to be a good solution in the medium to long term [46], but they are still far from affordable compared to HDD, and besides, databases still need to be optimized for them.

MapReduce has been criticized by some RDBMS proponents due to its low-level abstraction and lack of structure. But taking into account the different features and goals of relational databases and MapReduce, they can be seen as complementary rather than opposite models. So the most valid criticism is probably not related with the technical merits of MapReduce, but with the hype generated around it, which is pushing its use to solve problems for which it may not be the best solution.

2.1.3.2 *Distributed Key-value and Column-oriented DBMS*

Alternative database models such as Distributed Key-Value and Column-oriented DBMS are becoming more widely used for similar reasons as MapReduce. These two different approaches are largely inspired by Amazon's Dynamo [25] and Google's BigTable [19]. Key-value storage systems have properties of databases and distributed hash tables, while column-oriented databases serialize data by column, making it more suitable for analytical processing.

Both models depart from the idea of a fixed schema based structure, and try to combine the best of both worlds: distribution and scalability of systems like MapReduce with an higher and more database-oriented level of abstraction. In fact, some of the most popular data stores actually use or implement some sort of MapReduce. Google's BigTable, for instance, uses Google MapReduce to process data stored in the system, and other column-oriented DBMS such as CouchDB use their own implementations of MapReduce internally.

This kind of databases also mark a new trend and make it clear that the differences between traditional databases and MapReduce systems are blurring as developers try to get the best of both worlds.

2.1.3.3 *Grid Computing*

Like MapReduce, Grid computing services are also focused on performing computations to solve a single problem by distributing the work across several computers. But these kind of platforms often built on a cluster with a shared filesystem, which are good for CPU-bound jobs, but not good enough for data intensive jobs. And that's precisely one of the key differences between these kind of systems: Grid computing does not emphasize as much as MapReduce on data, and specially on doing the computation near the data.

Another distinction between MapReduce and Grid computing is the interface it provides to the programmer. In MapReduce the programmer is able to focus on the problem that needs to be solved since only the map and reduce functions need to be implemented, and the framework takes care of the distribution, communication, fault-tolerance, etc. In contrast, in Grid computing the programmer has to deal with lower-level mechanisms to control the data flow, check-pointing, etc. which makes it more powerful, but also more error-prone and difficult to write.

2.1.3.4 *Shared-Memory Parallel Programming*

Traditionally, many large-scale parallel applications have been programmed in shared-memory environments such as OpenMP. Compared to MapReduce, this kind of programming interfaces are much more generic and provide solutions for a wider variety of problems. One of the typical use cases of these systems are parallel applications that require some kind of synchronization (e.g. critical sections).

However, this comes at a cost: they may be more flexible, but the interfaces are also significantly more low-level and difficult to understand. Another difference between MapReduce and this model is the hardware for which each of this platform has been designed. MapReduce is supposed to work on commodity hardware, while interfaces such as OpenMP are only efficient in shared-memory multiprocessor platforms.

2.2 HADOOP

The code and experiments that are part of this project are all based on Hadoop. The decision to use Hadoop is primarily supported by the fact that Hadoop is not only the most complete Free software MapReduce implementation, but also one of the best implementations around.

Even though there are other open source MapReduce implementations, they either are still somewhat experimental or lack some component of the full platform (e.g. a storage solution). It is more difficult to compare to proprietary solutions, as most of them are not freely

available, but judging from the results of the Terasort benchmark [51] [52], Hadoop is able to compete even with the original Google MapReduce.

This section describes how MapReduce is implemented in Hadoop, and provides an overview of its architecture.

2.2.1 *Project and Subprojects*

Hadoop is currently a top level project of the Apache Software Foundation, a non-profit corporation that supports a number of other well-known projects such as the Apache HTTP Server.

Hadoop is mostly known for its MapReduce implementation, which is in fact a Hadoop subproject, but there are also other subprojects that provide the required infrastructure or additional components. The core of Hadoop upon which most of the other components are built is formed by the following subprojects:

COMMON The common utilities and interfaces that support the other Hadoop subprojects (configuration, serialization, RPC, etc.).

MAPREDUCE Software framework for distributed processing of large data sets on compute clusters of commodity hardware.

HDFS Distributed file system that runs on large clusters and provides high throughput access to application data.

The remaining subprojects are simply additional components that are usually used on top of the core subprojects to provide additional features.

2.2.2 *Cluster Overview*

A typical Hadoop MapReduce cluster is formed by a single master, also known as the *jobtracker*, and a number of slave nodes, also known as *tasktrackers*. The jobtracker is in charge of processing the user's requests, and distributing and scheduling the work on the tasktrackers, which are in turn supposed to execute the work they have been handed and regularly send status reports back to the jobtracker.

In the MapReduce context, a *job* is the unit of work that users submit to the jobtracker (Figure 2.1), and involves the input data as well as the `map()` and `reduce()` functions and its configuration. Jobs are divided into two different kinds of *tasks*, *map tasks* and *reduce tasks*, depending on the operation they execute. Tasktrackers control the execution environment of tasks and are configured to run up to a certain amount of *slots* of each kind. It defaults to 2 slots for map tasks and 2 slots for reduce tasks, but it can vary significantly depending on the hardware and the kind of jobs that are run in the cluster.

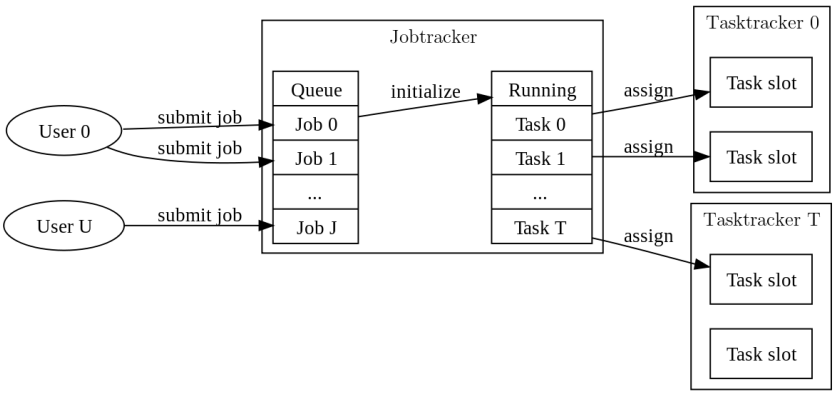


Figure 2.1: Job submission

Before assigning the first map tasks to the tasktrackers, the jobtracker divides the input data depending on its format, creating a number of virtual *splits*. The jobtracker then prepares as many map tasks as splits, and as soon as a tasktracker reports a free map slot, it is assigned one of the map tasks (along with its input split).

The master continues to keep track of all the map tasks, and once all of them have been completed it is able to schedule the reduce tasks[1]. Except for this dependency, for the jobtracker there is no real difference between kinds of tasks, so map and reduce tasks are treated similarly as the smallest scheduling unit.

Other than scheduling, the jobtracker must also make sure that the system is tolerant to faults. If a node fails or times out, the jobs the tasktracker was executing can be rescheduled by the jobtracker. Additionally, if some tasks make no apparent progress, it is also able to re-launch them as speculative tasks on different tasktrackers.

Note that Hadoop's master is not distributed and represents a single point of failure[2], but since it is aware of the status of the whole cluster, it also allows for some optimizations and reducing the complexity of the system[3].

2.2.3 *Storage with HDFS*

Hadooop MapReduce is designed to process large amounts of data, but it does so in a way that does not necessarily integrate perfectly well with previous tools, including filesystems. One of the character-

1 Actually, it is possible to start running reduce tasks before all map tasks are completed. For a more detailed explanation, see sections 2.2.4.

2 There have been numerous suggestions to provide a fault-tolerant jobtracker, such as Francesco Salbalori's proposal [69], but they have not made it into the official distribution.

3 Initial versions of Google's Filesystem and MapReduce are also known to have in masters their single point of failure to simplify the design, but more recent versions are reported to use multiple masters [67] in order to make them more fault-tolerant.

istics of MapReduce is that it moves computation to the data and not the other way around. In other words, instead of using an independent, dedicated storage, the same low-cost machines are used for both computation and storage. That means that the storage requirements are not exactly the same as for regular, general purpose filesystems.

The Hadoop Distributed File System [6] (HDFS) is designed to fulfill Hadoop's storage needs, and like the MapReduce implementation, it was inspired by a Google paper that described their filesystem [35]. HDFS shares many features with other distributed filesystems, but it is specifically conceived to be deployed on commodity hardware and thus it is supposed to be even more fault-tolerant.

Another feature that makes it different from other filesystems is its emphasis on streaming data and achieving high throughput rather than low latency access. POSIX semantics impose many requirements that are not needed for Hadoop applications, so in order to achieve its goals, HDFS relaxes some of the standard filesystem interfaces. Similarly, HDFS's coherency model is intentionally simple in order to perform as fast as possible, but everything comes at a cost: for instance, once a file is created, it is not possible to change it[4].

Like MapReduce, HDFS is also based on a client-server architecture. It consists of a single master node, also known as the *namenode*, and a number of slaves or clients known as *datanodes*. The namenode keeps all the metadata associated with the filesystem (permissions, file locations, etc.) and coordinates operations such as opening, closing or renaming. Datanodes are spread throughout the cluster and are responsible of storing the data, allowing read and write requests.

As in other general purpose filesystems, files in HDFS are split into one or more *blocks*, which is the minimum unit used to store files on datanodes and to carry out internal operations. Note that just like HDFS is designed to read and write very large files, its block size is likewise larger than the block size of other filesystems, defaulting to 64 MB. Also, to ensure fault tolerance, files have a replication factor, which is used to enforce the number of copies of each block available in the cluster.

In order to create a new file, the client first requests it to the namenode, but upon approval it writes directly to the datanodes (Figure 2.2). This process is handled by the client and is transparent for the user. Similarly, replication is coordinated by the namenode, but data is directly transferred between datanodes. If a datanode fails or times out, the namenode goes through all the blocks that were stored in that datanode, issuing replication requests for all the blocks that have fallen behind the desired replication ratio.

4 There have been patches to enable appends in previous versions of HDFS, and its support was briefly included in 0.19.0 before dropping it in the 0.19.1 release due to technical problems. At the moment of writing this document, there is ongoing work to support file appends again in the upcoming 0.21 release.

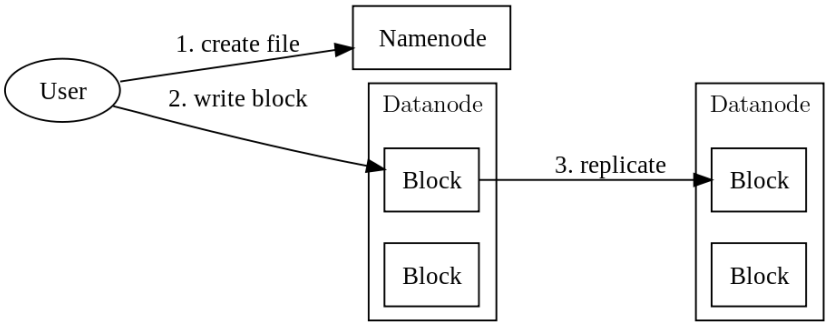


Figure 2.2: HDFS file creation

2.2.4 *Dataflow*

The previous sections introduced how MapReduce and the filesystem work, but one of the keys to understanding Hadoop is to know how both systems are combined and how data flows from the initial input to the processing and final output.

Note that although the MapReduce model assumes that data is available in a distributed fashion, it does not directly deal with pushing and maintaining files across the cluster, which is the filesystem's job. A direct advantage of this distinction is that Hadoop's MapReduce supports a number of filesystems with different features. In this description, though, as well as in the remaining chapters of this document, the cluster is assumed to be running HDFS (described in the previous section 2.2.3), which is the most widely used filesystem.

MapReduce is able to start running jobs as soon as the required data is available in the filesystem. First of all, jobs are initialized by creating a series of map and reduce tasks. The number of map tasks is usually determined by the number of splits into which the input data is divided. Splitting the input is what makes it possible to parallelize the map phase and can have a great impact on the performance, so splits can also be thought of as the first level of granularity of the system, and it also shows how the filesystem and MapReduce are integrated. For instance, if the input consists of a single 6.25 GB file in an HDFS filesystem, using a block size (`dfs.block.size`) of 64 MB and the default input format, the job will be divided into 1000 map tasks, one for each split.

Map tasks read its share of the input directly from the distributed filesystem, meaning they can read either locally if the data is available, or remotely from another host if it is not (Figure 2.3). While reading and processing the input, the partial output is continuously written to a circular memory buffer. As can be observed in Figure 2.4, as soon as the buffer reaches a certain threshold (defined by `io.sort.spill.percent`, defaults to 80%), its contents are sorted and flushed to a temporary file in the local disk. After reading the input, if

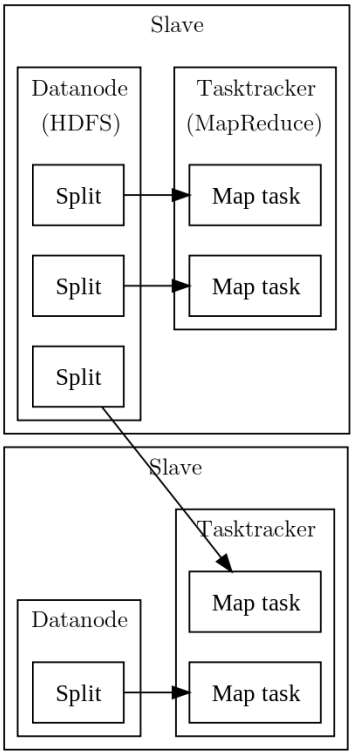


Figure 2.3: Local and remote reads from HDFS to MapReduce

there is more than one temporary file, the map task will merge them and write the merged result again to disk. Optionally, if the number of spills is large enough, Hadoop will also perform the combine operation at this point in order to make the output smaller and reduce bandwidth usage.

Note that in the end it is always necessary to write the map's result to disk even if the buffer is not completely filled: map tasks run on its own JVM instance and are supposed to finish as soon as possible and not wait indefinitely for the reducers. So after writing to disk, the map's partial output is ready to be served to other nodes via HTTP.

The number of reduce tasks is determined by the user and the job's needs. For example, if a job requires global sorting, a single reducer may be needed[5]. Otherwise, any number of reduces may be used: using a larger number of reducers increases the overhead of the framework, but can also help to improve the load balancing of the cluster.

Reduce tasks are comprised of three phases: copy, sort and reduce. Even though reduce tasks cannot be completed until all map tasks are, it is possible to run the first phase of reduce tasks at the same time as map tasks. During the copy phase (also known as shuffle),

5 For some kind of problems there may be more efficient options such as chaining multiple jobs, using the output of one job as the input for the next one.

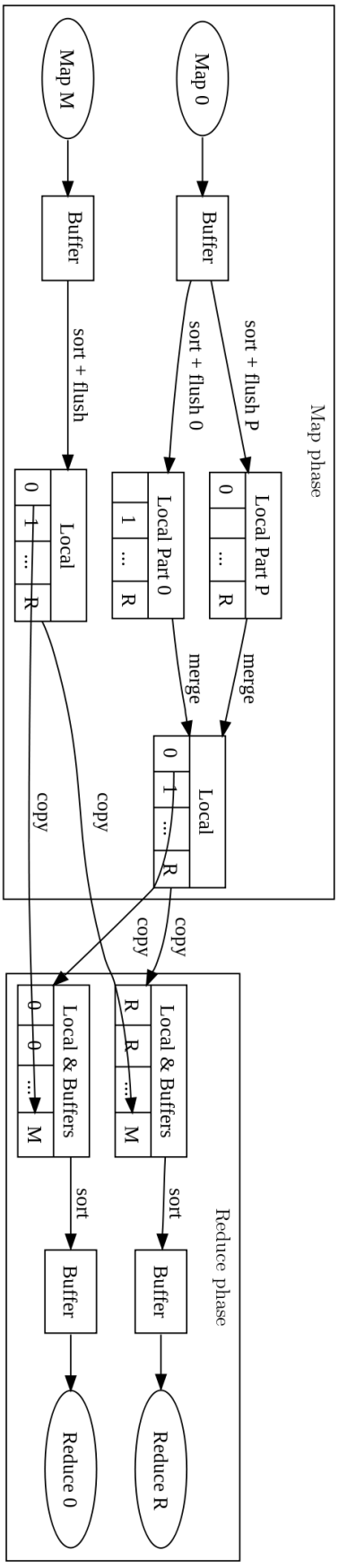


Figure 2.4: Hadoop dataflow

reduce tasks request their partitions of data to the nodes where map tasks have already been executed, via HTTP. As soon as data is copied and sorted in each reducer, it is passed to the `reduce()` function, and its output is directly written to the distributed filesystem.

2.3 SCHEDULING CONCEPTS

This section provides an overview of fundamental concepts behind traditional scheduling of parallel jobs, and how they relate and apply to MapReduce environments.

2.3.1 *Parallel Job Scheduling*

At the single-machine level, operating systems scheduling traditionally deals with the execution of threads on the machine's processor or processors.

In this context, the problem of scheduling multiple parallel jobs and their constituent threads in order to minimize some given metric has been studied extensively [47, 32, 31]. Depending on the way in which resources are shared between jobs, scheduling policies can be classified as either single-level (time-sharing) or two-level (space-sharing) [29]. The former approach schedules threads directly on processors, while the latter decouples the scheduling into two decisions: first, allocating processors to jobs; and second, selecting which threads of each job to run.

In addition to considering the way in which resources are shared, different schemes intended to schedule parallel jobs on parallel machines also differ depending on the capabilities and characteristics of the systems and its applications. Feitelson classified parallel jobs into rigid, moldable, evolving, or malleable [29]. *Rigid* jobs require a certain predefined and fixed number of processors. *Moldable* jobs are able to run with any number of processors, but the number does not change at runtime. *Evolving* jobs can be thought as having a number of phases, each with different requirements in terms of number of processors. *Malleable* jobs can be executed with a dynamic number of processors, and may be adjusted at runtime.

2.3.2 *Cluster Scheduling*

As architectures evolved, clusters of connected computers working together as a single system became more popular for HPC and other kinds of workloads.

Akin to single-machine parallel scheduling, cluster-level scheduling makes use of multiple computers to execute parallel applications. Therefore, many of the strategies used for parallel job scheduling in parallel processors can be and have been adapted to clusters [32].

Access to cluster infrastructures was originally dominated by batch systems. Similar to parallel job scheduling, schedulers in batch systems decide which jobs are executed, as well as when and where to run them. A standard scheduling algorithm in this kind of environment consists of following a plain First-Come-First-Serve approach, with some variation of backfilling, allowing some jobs in the waiting queue to run ahead of time so as to avoid resource fragmentation and improve the utilization of the system. Backfilling policies have been widely studied in the literature [31]: from simple conservative backfilling that guarantees reservations made by previously submitted jobs [74], to more dynamic and aggressive approaches that may have an impact on previously submitted jobs [48].

2.3.3 *Grid Scheduling*

Grid computing originated in the nineties as a metaphor for easily available *computational* power, and became canonical when Ian Foster et al coined the term [34].

Like regular clusters, the grid can also be thought of as a set of computational resources. However, while clusters are generally tightly coupled and share a centralized job management and scheduling, the grid is usually characterized as more loosely coupled, heterogeneous, and geographically distributed, which often leads to more decentralization and imposes additional challenges.

Previously studied techniques for parallel job scheduling have been adapted to grid systems [32], but grid scheduling generally deals with an additional set of problems and constraints [87]. Grid scheduling needs to take into account the fact that resources may have different owners with different goals, and may involve a meta-scheduler and additional steps such as resource discovery and brokering since the grid does not control resources directly [27]. The characteristics of the grid also lead to changes in the scheduling techniques that are used. For instance, scheduling batch jobs on clusters often relies on user estimates, which are feasible in homogeneous environments; however, the complexity of the grid requires different approaches, often based on prediction techniques, since the user does not have enough information to provide good estimates.

2.3.4 *MapReduce Scheduling*

The MapReduce model is designed to support clusters of computers running multiple concurrent jobs, which are submitted to a centralized controller which deals with both job allocation and task assignment.

From a scheduling perspective, MapReduce jobs are formed by two different phases: map and reduce. A key feature of MapReduce and

its parallel structure is a consequence of how each one of these phases can be split into smaller tasks. Roughly speaking, each phase is composed of many atomic tasks that are effectively independent of each other and therefore can be simultaneously executed on an arbitrary number of multiple hosts in the cluster (or slots, in MapReduce terminology). Consider an environment that provides a total of N hosts. In scheduling theory, a job is said to be *parallelizable* if it can be performed on an arbitrary number of hosts $n \leq N$ simultaneously, with an execution time $F(n)$ that depends on the number of hosts allocated. F is the speedup function: if a given job is allocated more resources, it will complete in a smaller amount of time. Both MapReduce phases, map and reduce, can be approximated as parallelizable, and in particular malleable [85].

The problem of scheduling multiple malleable jobs has been studied to solve different metrics, such as response times or makespan [17], but these are generally applied to simple jobs consisting of a single kind of task [80, 81]. In MapReduce, however, a malleable approach can't be applied globally since jobs are actually composed of two malleable sub-jobs or phases, and there is also a clear precedence between them: tasks of the map phase are to be executed before tasks of the reduce phase.

In addition to the issue of applying regular malleable scheduling, MapReduce also presents certain unique characteristics. The task granularity in MapReduce is not as fine-grained as the granularity usually found in operating systems scheduling, where the allocation time if fixed and known precisely. Therefore, MapReduce tasks are not assigned nor freed perfectly: certain running tasks may take slightly less or more time than expected to be executed, and so new the tasks tasks may also have to wait to run.

Traditional scheduling methods can also perform poorly in MapReduce due to the need of data locality and running the computation where the data is. Scheduling based on space-slicing is often associated with exclusive allocation of processors to jobs, and there is usually a notion of affinity to avoid costly context switching. Similarly, traditional cluster schedulers give each user a fixed set of machines. This kind of approach is actually detrimental to MapReduce jobs since it it prevents the execution of tasks where the data is available [91], and may degrade its performance since all data is distributed across all nodes and not in a subset of them. Sharing the data in the MapReduce cluster also leads to data consolidation, avoiding costly replication of data across private clusters and allowing queries across disjoint data sets. Some grid schedulers like Condor [77] support some kind of locality constraints, but they are focused on geographically distributed sites rather than the machine-level locality desired for the data-intensive MapReduce workloads.

The MapReduce schedulers described in this thesis target a highly dynamic environment as that described in [18], with multiple jobs and multiple users, in which MapReduce workloads may share physical resources with other workloads. New jobs can be submitted at any time with different priorities and needs: from small, almost interactive executions to very long programs that may take hours to complete.

In this scenario, the actual amount of resources available for applications can vary over time depending on the workload. The MapReduce scheduler must be able to respond with an online algorithm to the varying amounts of resources and jobs. User-defined completion time goals are used as a mechanism to prioritize jobs, but unlike real-time systems scheduling [71], these are soft-deadlines that simply guide the execution.

2.4 HARDWARE HETEROGENEITY

Current research trends [72] show that next generation data centers will contain a remarkable degree of heterogeneity of nodes (i.e. the RoadRunner [49] cluster, composed of Opterons and Cell/BE blades), in an attempt to improve data center power efficiency and performance. Such heterogeneity, involving generic and specialized processors co-existing in the same data center and performing differentiated tasks, hinders the efficient use of node resources in a convenient way. In this scenario, the new challenge is to transparently exploit these heterogeneous resources, such that applications can (where possible) experience improved performance when executed on this kind of specialized system.

The work presented in this thesis takes advantage of a prototype that extends the Hadoop runtime to access the capabilities of underlying hardware accelerators [15]. The architecture of the prototype has two main components. (see Figure 2.5). The first component is based on Hadoop and it partitions the data and assigns a piece of work to each node in the cluster. The second component implements a second-level partition of the data (intra-node distribution), and does the actual computation. The processing routine executed by each node (the *map()* function) invokes the second component of our prototype using the Java Native Interface [73]. Notice that this environment does not add complexity to the task of programming applications that can exploit hardware accelerators: programmers only have to provide a *map()* function that is able to exploit specialized hardware.

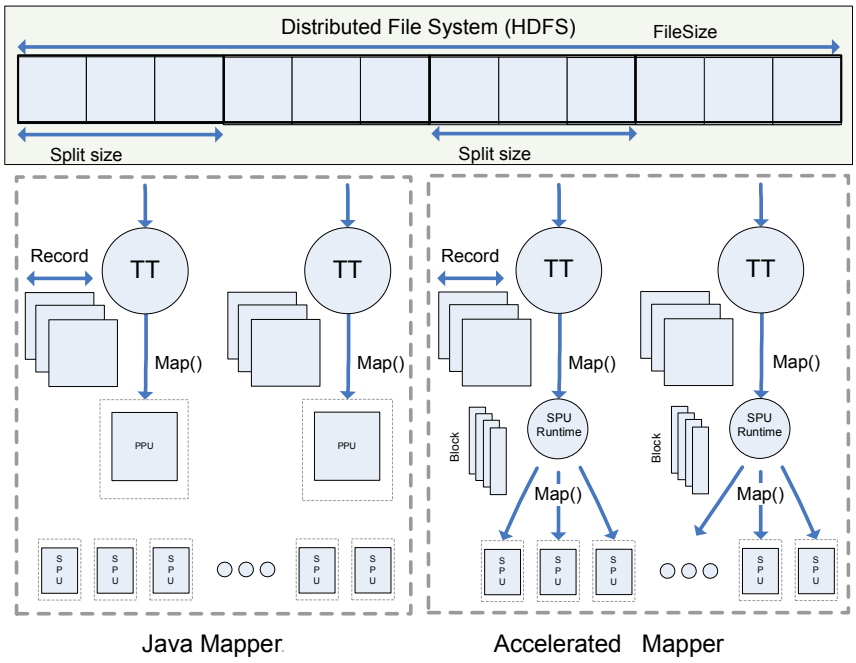


Figure 2.5: Architecture of the system: 2 levels of parallelism

2.5 DATA STORES

2.5.1 *Cassandra*

Apache Cassandra [45] is a distributed database management system initially developed by Facebook for internal usage and later released as an open source project. Cassandra inherits its data model from Google's BigTable [19], and its replication mechanism and distribution management from Amazon's Dynamo [25]. We use Cassandra as an example of a widely used key-value store known for its scalability and support for tunable consistency.

Cassandra's data model is schema-free, meaning there is no need to define the structure of the data in advance. Data is organized in *column families*, which are similar to tables in a relational database model. Each column family contains a set of *columns*, which are equivalent to attributes, and a set of related columns compose a *row*. Each row is identified by a *key*, which are provided by applications and are the main identifier used to locate data, and also to distribute data across nodes. Cassandra does not support relationships between column families, disregarding foreign keys and join operations. Knowing this, the best practice when designing a data model is to keep related data in the same column family, denormalizing it when required.

The architecture of Cassandra is completely decentralized and peer-to-peer, meaning all nodes in a Cassandra cluster are equivalent and provide the same functionality: receive read and write requests, or

forward them to other nodes that are supposed to take care of the data according to how data is partitioned.

When a read request is issued to any target node, this node becomes the proxy of the operation, determines which nodes hold the data requested by the query, and performs further read requests to get the desired data directly from the nodes that hold the data. Cassandra implements automatic partitioning and replication mechanisms to decide which nodes are in charge of each replica. The user only needs to configure the number of replicas and the system assigns each replica to a node in the cluster. Data consistency is also tunable by the user when queries are performed, so depending on the desired level of consistency, operations can either return as soon as possible or wait until a majority or all nodes respond.

2.5.2 *Isolation and Consistency Levels*

The ultimate goal of current distributed key-value stores such as Cassandra [45] is similar to other database systems, reading and writing data operations, but with a stronger focus on adapting to the increased demands of large-scale workloads. While traditional databases provide strong consistency guarantees of replicated data by controlling the concurrent execution of transactions, Cassandra provides tunable consistency in order to favour scalability and availability. While there is no tight control of the execution of concurrent transactions, Cassandra still provides mechanisms to resolve conflicts and provide durability even in the presence of node failures.

Traditionally, database systems have provided different isolation levels that define how operations are visible to other concurrent operations. The ANSI SQL standard defines 4 isolations levels, which can be classified depending on the anomalies that they exhibit, as shown in Table 2.1.

LEVEL	READ PHENOMENA
Read Uncommitted	Dirty reads
Read Committed	Non-repeatable reads
Repeatable Reads	Phantom reads
Serializable	-

Table 2.1: Isolation Levels as defined by the ANSI/ISO SQL Standard

Reading the same table twice within a transaction will have a different outcome depending on the isolation level. Under the Read Uncommitted level, transactions are exposed to *dirty reads*, meaning a transaction may be reading updates that have not been committed yet. The next level, Read Committed, does not allow reading uncom-

mitted updates, but it still allows *non-repeatable reads*: a second read within a transaction may return data updated and committed by another transaction. The Repeatable Read level further guarantees that rows that have been read remain the same within a transaction, but it does not deal with range queries, which can lead to *phantom reads* (e.g. when there are additions). Finally, Serializable provides the highest isolation level and guarantees that the outcome of executing concurrent transactions is the same as if they were executed serially, avoiding any kind of read anomaly.

Standard ANSI SQL isolation levels have been criticized as too few [16], but in addition to standard ANSI SQL, other non-standard levels have been widely adopted by database systems. One such level is Snapshot Isolation, which guarantees that all reads made within a transaction see a consistent version of the data (a *snapshot*). While Snapshot Isolation does not exhibit any of the read anomalies defined by standard SQL, it still does not provide as much isolation as the Serializable level since it can be exposed to write anomalies instead. For instance, two transactions reading overlapping data can make disjoint concurrent updates (also known as *write skew*), which would not be possible under the Serializable isolation level.

3

SCHEDULING WITH TIME CONSTRAINTS

3.1 INTRODUCTION

Cloud computing has dramatically transformed the way many critical services are delivered to customers (for example, the Software, Platform, and Infrastructure as a Service paradigms), and at the same time has posed new challenges to data centers. The result is a complete new generation of large scale infrastructures, bringing an unprecedented level of workload and server consolidation, that demand new programming models, management techniques and hardware platforms. At the same time, it offers extraordinary capacities to the mainstream market, thus providing opportunities to build new services that require large scale computing. Therefore, data analytics is one of the more prominent fields that can benefit from next generation data center computing.

The intersection between cloud computing and next generation data analytics services [2] points towards a future in which massive amounts of data are available, and users will be able to process this data to create high value services. Consequently, building new models to develop such applications, and mechanisms to manage them, are open challenges. An example of a programming model especially well-suited for large-scale data analytics is MapReduce [24], introduced by Google in 2004.

MapReduce workloads usually involve a very large number of small computations executing in parallel. High levels of computation partitioning, and relatively small individual tasks, are a design point of MapReduce platforms. In this respect, MapReduce workloads are closer to online web workloads than to single-process batch jobs. And while it was originally used primarily for batch data processing, its

use has been extended to shared, multi-user environments in which submitted jobs may have completely different priorities. This change makes scheduling even more relevant. Task selection and slave node assignment govern a job's opportunity to progress, and thus influence job performance.

One of the design goals of the MapReduce framework is to maximize data locality across working sets, in an attempt to reduce network bottlenecks and increase (where possible) overall system throughput. Data locality is achieved when data is stored and processed on the same physical nodes. Failure to exploit locality is one of the well-known shortcomings of most multi-job MapReduce schedulers, since placing tasks from different jobs on the same nodes will have a negative effect on data locality.

At the same time, there is a trend towards the adoption of heterogeneous hardware ([72, 49]) and hybrid systems [21] in the computing industry. Heterogeneous hardware (mixing generic processors with accelerators such as GPUs or the SPUs in the Cell/BE [37] processor) will be leveraged to improve both performance and energy consumption, exploiting the best features of each platform. For example, a MapReduce framework enabled to run on hybrid systems [15] has the potential to have considerable impact on the future of many fields, including financial analysis, healthcare, and smart cities-style data management. MapReduce provides an easy and convenient way to develop massively distributed data analytics services that exploit all the computing power of these large-scale facilities. Huge clusters of hybrid many-core servers will bring workload consolidation strategies one step closer in future data centers.

The main contribution described in this chapter is a scheduling algorithm and technique for managing multi-job MapReduce workloads that relies on the ability to dynamically build performance models of the executing workloads, and uses these models to provide dynamic performance management. At the same time, it observes the particulars of the execution environment of modern data analytics applications, such as hardware heterogeneity and distributed storage. Beyond the formulation of the problem and the description of the scheduling technique, a prototype (called Adaptive Scheduler) has been implemented and tested on a medium-size cluster. The experiments study, separately, the following topics:

- The scheduler's ability to meet high level performance goals guided only by user-defined completion time goals;
- The scheduler's ability to favor data-locality; and
- The scheduler's ability to deal with hardware heterogeneity, which introduces hardware affinity and relative performance characterization for those applications that can benefit from executing on specialized processors.

The remaining sections of the chapter are structured as follows. Section 3.2 summarizes the scheduling approach. Section 3.3 describes the method by which priorities are assigned to jobs (which is the core of the scheduling algorithm), and Section 3.4 describes the three different allocation policies implemented as part of the scheduler presented in this chapter. Section 3.5 presents experiments to support the evaluation of the scheduler. Finally, Section 3.6 discusses related work, and Section 3.7 provides a summary of the chapter.

3.2 SCHEDULING PRINCIPLES

The main contribution of this chapter is to present the design, implementation and evaluation of the Adaptive Scheduler, a performance-driven MapReduce scheduler that provides integrated management of next generation data centers, considering data-locality of tasks and hardware heterogeneity.

The task scheduling approach presented here enables MapReduce runtimes to dynamically allocate resources in a cluster of machines based on the observed progress achieved by the various jobs, and the completion time goal associated with each job. A necessary component of such a system is an estimator that maps the resource allocation for a job to its expected completion time. Such an estimator can easily be implemented, provided that information about the total amount of computation to be performed by a job is known in advance. One way to provide this information would be to derive it from prior executions of the job: however, this approach is neither practical (as it cannot be guaranteed that prior executions of the job exist), nor accurate (as prior executions of the job were likely performed over a different data set and may therefore have completely different characteristics).

This thesis follows a different approach which is to dynamically estimate the completion time of a job during its execution. In doing so, the scheduler takes advantage of the fact that MapReduce jobs are a collection of a large number of smaller tasks. More specifically, the hypothesis in which the scheduler is based is that from a subset of tasks that have completed thus far, it is possible to predict the properties of remaining tasks. It should be noted that MapReduce tasks may vary widely in their execution characteristics depending on the data set they process. Hence, while this estimation technique is not expected to provide accurate predictions all the time, but when combined with dynamic scheduling it will permit fair management of the completion times of multiple jobs.

The main goal and metric of the scheduler is to minimize the distance to the deadline of the jobs. Completion time estimates and performance goals are the basis for dynamically calculate the priority of each job. Two additional extensions to this policy have also designed

and implemented, considering two different kinds of job affinity: *data affinity* and *hardware affinity*. Recall, however, that meeting the performance goal of each job remains the primary criteria that guides scheduling decisions: affinities are only favored when possible.

For the data affinity extension, data locality is taken into account before making scheduling decisions. Data locality is achieved when data is stored and processed on the same physical nodes. This work proposes a policy that improves the percentage of local tasks by delaying the execution of remote tasks while the performance goal of the job is still reachable. Although other schedulers delay the execution of remote tasks to favor data locality ([90, 91]), this proposal is the only one that also considers the completion time goal of the applications before delaying.

The hardware-affinity extension enables the scheduler to deal with heterogeneous hardware (general purpose cores and specialized accelerators such as GPUs) and thus to exploit the multi-level parallelism available in next generation heterogeneous clusters. This feature will allow the scheduler to determine, at runtime, if some tasks are 'accelerable': that is, if they can benefit from executing on nodes enabled with accelerators. These accelerable tasks will, if possible, be assigned to nodes with accelerators. Once again, meeting the performance goal for all kind of tasks is still the main goal of the scheduling mechanism. This proposal represents the first MapReduce scheduler that is able to manage heterogeneous hardware while observing jobs' completion time goals. This extension focuses on just one hardware heterogeneity dimension. However, it is feasible to add more heterogeneity considerations to the scheduling criteria. For example, [63] describes an extended task scheduling approach to consider the amount of resources available at each node together with the particular resource requirements of each application.

The scheduling technique targets a highly dynamic environment, such as that described in [18], in which new jobs can be submitted at any time, and in which MapReduce workloads share physical resources with other workloads, both MapReduce and not. Thus, the actual amount of resources available for MapReduce applications can vary over time. The dynamic scheduler introduced in this chapter uses the completion time estimate for each job given a particular resource allocation to adjust the resource allocation to all jobs. If there are enough resources to meet all goals, the remaining resources are fairly distributed among jobs. The minimum unit of resource allocation is the *slot*, which corresponds to a worker process created by a TaskTracker.

The scheduler can be considered pre-emptive in that it can interrupt the progress of one job in order to allocate all of its resources to other jobs with higher priority; but it does not interrupt tasks that are already executing. Interrupting executing tasks could be beneficial in

the case of, for example, reduce tasks with a long copy phase: this issue is part of the future work of this thesis.

3.3 PERFORMANCE ESTIMATION

This section presents the method to dynamically estimate job performance and thus calculate its priority when scheduling in a shared, multi-job environment.

3.3.1 *Problem Statement*

We are given a set of jobs M to be run on a MapReduce cluster. Each job m is associated with a completion time goal, T^m_{goal}. The Hadoop cluster includes a set of TaskTrackers TT, each TaskTracker (TT_t) offering a number of execution slots, s_t, which can host a task belonging to any job. A job (m) is composed of a set of tasks. Each task (t^m_i) takes time α^m_i to be completed, and requires one slot to execute.

A MapReduce job has two different types of tasks, depending on the execution phase of the job: map tasks and reduce tasks. In a general scenario, map tasks length are regular and differ from reduce tasks length. In order to get suitable accuracy in the job performance estimation, we estimate the performance for each job phase, map and reduce, separately. And we decide the number of slots to allocate considering the current execution phase and the completion time goal of the phase $T^{m,p}_{goal}$, which is calculated based on the completion time goal of the job and the time required for executing each phase. For the sake of clarity, in the description of the performance model and in the description of the allocation algorithms, we will refer to tasks and completion time goal, without specifying the involved execution phase.

The set of tasks for a given job m can be divided into tasks already completed (C_m), not yet started (U_m), and currently running (R_m). We also use $C_{m,t}$ to denote the set of tasks of job m already completed by TT_t.

3.3.2 *Modelling Job Performance*

Let μ_m be the mean completed task length observed for any running job m, denoted as $\mu_m = \frac{\sum_{i \in C_m} \alpha^m_i}{|C_m|}$. Let μ^t_m be the mean completion time for any task belonging to a job m and being run on a TaskTracker TT_t. Notice that as the TaskTrackers are not necessarily identical, in general $\mu_m \neq \mu^t_m$. When implementing a task scheduler which leverages a job completion time estimator, both μ_m and μ^t_m should be considered. However, in the work presented in this chapter, only μ_m is considered, i.e., all μ^t_ms are presumed equal. Three reasons have motivated

this decision: 1) a design goal is to keep the scheduler simple, and therefore all slots are considered identical. Under this assumption, estimating the allocation required by each job given its completion time goal is an easy task that can be performed with cost $O(M)$. If the differences between TaskTrackers are taken into account, the cost of making the best allocation for multiple jobs could grow to be exponential. 2) The scenario in which task scheduling occurs is highly dynamic, and thus the scheduling and completion time estimate for each job is updated every few minutes. Therefore, a highly accurate prediction provides little help when scheduling tasks in a scenario in which external factors change the execution conditions over time. The approach is focused on dynamically driving job allocation under changing conditions. And 3) the completion time estimate for a job m can only benefit from having information relative to a particular TaskTracker if at least one task that belongs to the job has been scheduled in that TaskTracker. In practice, it is likely that each job will have had tasks scheduled on only a small fraction of the TaskTrackers.

For any currently executing task t_i^m we define β_i^m as the task's elapsed execution time, and δ_i^m as the remaining task execution time. Notice that $\alpha_i^m = \beta_i^m + \delta_i^m$, and that δ_i and α_i^m are unknown. Our completion time estimation technique relies on the assumption that, for each on-the-fly task t_i^m, the observed task length α_i^m will satisfy $\alpha_i^m = \mu_m$, and therefore $\delta_i^m = \mu_m - \beta_i^m$.

3.4 ALLOCATION ALGORITHM & EXTENSIONS

In order to complete the implementation of the scheduler it is necessary to provide an allocation algorithm that assigns free slots to jobs based on their priority and affinity features.

Jobs are organized in an ordered queue based on their priority. The current implementation updates the priority queue on every call to the scheduler, which has a cost of $O(n \log n)$, where n is the number of running jobs. This has proven adequate for testing purposes and keeps the prototype simple. However, as the queue may not change much between updates, and the number of available slots is usually small, this approach results in unnecessary work. We plan to improve efficiency by updating the queue in a background process.

In the event that several jobs have the same priority, one of them is chosen arbitrarily. This is not a problem as, once a slot is allocated to one of these jobs, its priority will decrease, and the next slot will be allocated to one of the other jobs that previously had the same priority. When two jobs that have already missed their deadlines compete for resources, the scheduler fairly equalizes their expected completion times with respect to their goals. When there are slots that are *not needed* to satisfy the completion time goal of all the jobs, the scheduler allocates excess slots to jobs with the highest priority. Priorities

are updated after each allocation, so the process does not necessarily assign all excess slots to the same job.

In the following subsections we present and evaluate three different allocation algorithms:

- Basic Adaptive Scheduling: does not consider any kind of job affinity: the performance goals of the jobs are the only guide for scheduling decisions (see section 3.4.1);
- Adaptive Scheduling with Data Affinity: data locality considerations complement the information about job priority when making scheduling decisions (see section 3.4.2); and
- Adaptive Scheduling with Hardware Affinity: hardware affinity complement the information about job priority when making scheduling decisions (see section 3.4.3).

3.4.1 *Basic Adaptive Scheduling*

The priority of each job is calculated based on the number of slots to be allocated concurrently to that job over time so as to reach its completion time goal. For such purposes, we still need to estimate the amount of pending work for each job, assuming that each allocated slot will be used for time μ_m. Such estimation needs to consider both the tasks that are in the queue waiting to be started, and those that are currently on execution. Based on these two parameters, we propose that the number s^m_{req} of slots to be allocated in parallel to a job m can be estimated as:

$$s^m_{req} = \frac{\left(\frac{\sum_{i \in R_m} \delta^m_i}{\mu_m} + |U_m|\right) * \mu_m}{T^m_{goal} - T_{curr}} - |R_m| \tag{1}$$

where T^m_{goal} is the completion time goal for job m, and T_{curr} is the current time. Therefore, the order in queue is defined by s^m_{req}, dynamically calculated for each job.

The scheduling policy must consider some special jobs which get the highest priority in the system. First, jobs that have already missed their deadline. For such a job, the scheduler tries to at least complete it as soon as possible, which helps avoid job starvation. Second, jobs with no completed task. Immediately after a job is submitted, there is no data available and it is not possible to estimate the required slots or the completion time (if there is more than one such job, the oldest one comes first). In summary, the priority of the job is calculated as follows: First, jobs that have already missed the deadline. Second, recently submitted jobs for which there is no available data. Finally, executing jobs based on their s^m_{req}. The Adaptive Scheduler's code can be found at [57].

3.4.2 *Adaptive Scheduling with Data Affinity*

To enable the Basic Adaptive Scheduling with a mechanism to improve data locality, we defer, if possible, the execution of those tasks that are assigned to run on TaskTrackers with no local data, thus allowing other jobs, possibly with local tasks, to use that slot. The decision to defer remote tasks is made dynamically on a per-job basis, each time the next task to schedule happens to be remote. The computation is guided by two parameters: the current distance of the job from its performance goal, and the maximum number of delays that the user agrees to allow for each task of the job. When the next task to schedule cannot execute locally to its data, the Adaptive Scheduler uses the estimated number of required slots to determine whether remote tasks may be avoided or not: if the job has already been assigned enough slots to meet the deadline, then remote tasks will be deferred until the maximum number of delays per task is reached. In situations when no tasks with local data in the TaskTracker are ready to run, the JobTracker will decide to start a task that needs remote data.

This approach is completely different and unique to the Adaptive Scheduler since it is based on the job's performance goal. One of the advantages it provides over other approaches is that it allows the execution of remote tasks when it is needed to meet the performance goal of jobs, instead of just avoiding all remote executions.

The Fair Scheduler ([90, 91]) also aims to improve data locality of jobs by delaying the execution of remote tasks. However, this scheduler uses a fixed maximum time delay defined statically, which applies to all running jobs without considering the distance of each job from its performance goal, and thus it can cause jobs to miss their goals. For example, let's consider a workload composed of two applications: one of them with a sufficiently relaxed completion time goal to be ignored during the deferring decision, and the other one with a tight completion time goal that requires the scheduler to execute some remote task in order to meet its completion time goal. In this situation, taking a per-application delay decision enables the system to get the maximum data locality factor for each application without missing their completion goal time. For the first application, the scheduler defers all remote tasks; and for the second one it stops delaying remote tasks when the completion time goal gets compromised.

Even for executions that have very relaxed job performance goals, and for which meeting performance goals should not be an issue, defining the locality delay in a static and global fashion is not desirable in a system that must handle all kinds of workloads and data placement patterns.

For the sake of clarity, consider the following two scenarios that illustrate the need for a dynamic approach to defining per-application locality delay. Assume that, in both scenarios, the performance goals of the jobs is sufficiently relaxed to be ignored during the deferring decision. In the first case, imagine an application for which all data blocks are stored on the same DataNode, and thus, to achieve maximum locality, the job's tasks must be run sequentially, and the locality delay must be set to an extremely high value; in a second case, imagine the same application having a replica of all data blocks in each node of the cluster, and thus, a locality delay has no effect.

3.4.3 *Adaptive Scheduling with Hardware Affinity*

Scheduling jobs that contain both accelerable and non-accelerable MapReduce task implementations requires the scheduler to keep track of different kinds of TaskTrackers depending on the hardware characteristics of the nodes in which they are running. Whenever a job is deployed with an accelerator-based version of its code and one of the tasks for that job is scheduled to run on an accelerator-enabled TaskTracker, the code that will run in that node will be the accelerator-specific version. Otherwise, the regular non-accelerated Java version of the code will be run on the node.

TaskTrackers are configured and grouped into pools of machines with different features: TaskTrackers running on regular machines (non accelerator-enabled) are included in the regular pool P_{reg}, while TaskTrackers running on accelerated machines are included into the accelerated pool denoted by P_{acc}. These pools are used not only to favor the affinity and execute accelerable tasks on accelerator-enabled nodes, but also to detect whether new jobs may benefit from accelerators or not. During an initial *calibration* stage, immediately after jobs are submitted, the scheduler first makes sure to execute at least one map task on each pool. Then, as soon as these initial tasks are completed, the scheduler decides whether or not a job will benefit from running on machines from that pool by comparing the observed elapsed task times on accelerated TaskTrackers (μ^m_{acc}) with those obtained on the regular pool (μ^m_{reg}). Recall that some jobs that are I/O bound may not clearly benefit from task acceleration even if their code can be run on an accelerator. In that case, providing affinity to accelerator-enabled nodes would bring no benefit and could even result in competition for accelerators with other jobs that in fact could take advantage of them. Therefore, only in the case that the speedup obtained when running on one of the accelerated pools (μ^m_{reg}/μ^m_{acc}) passes a certain configurable threshold will the job be marked as accelerable, and will preferably run on TaskTrackers from P_{acc}. Otherwise it will be marked as regular and will be preferably executed on P_{reg}. In this version of the scheduler we are considering accelerable

map tasks only, but it would be straightforward to extend it for accelerable reduces as well. The main changes would involve a calibration stage for reduces.

Other than detecting accelerable jobs, the scheduler itself still assigns resources depending on job priority, which is in turn primarily based on the need of task slots to meet the completion time goal (s^m_{req}). However, this estimation is now slightly different: for accelerable jobs, only the mean time of tasks executed on accelerator-enhanced hardware are observed:

$$s^m_{req,acc} = \frac{\left(\frac{\sum_{i \in R^m_{acc}} \delta^m_i}{\mu^m_{acc}} + |U^m_{acc}|\right) \times \mu^m_{acc}}{T^m_{goal} - T_{curr}} - |R^m_{acc}| \tag{2}$$

It should be noted, though, that this approach results in another issue: even though jobs can be easily prioritized within each pool, the scheduler still needs a global prioritization if there are not enough resources to meet the completion time goals. This is accomplished by normalizing the need of slots of accelerable jobs ($s^m_{req,extra}$) depending on the observed speedup as well as the capacity of the accelerated pool:

$$s^m_{req,extra} = \left(\frac{\mu^m_{reg}}{\mu^m_{acc}}\right) \times \left(s^m_{req,acc} - \sum_{t \in P_{acc}} s_t\right) \tag{3}$$

For the sake of clarity, take for instance a cluster with 10 slots running on accelerator-enabled machines, and a job whose map tasks take 50*s* to complete on an accelerator and 100*s* on a regular machine. Since it is accelerable, if the job needs 10 or less slots to meet the completion time goal, the scheduler will try to schedule it on accelerators only. However, if the job is missing its goal and accelerable nodes are not enough, the number of required slots will change accordingly: an estimation of 15 accelerated slots to meet the completion goal will be normalized to 10 accelerated slots and 10 regular slots ($(100/50) \times (15 - 10)$).

This way accelerable jobs are prioritized over regular ones with a similar need for resources, but the latter are not excluded from running if they have a much tighter completion time goal. Similarly, if there are not enough resources to complete an accelerable job on time, these are able to request additional non-accelerated slots, considering the performance difference among the execution on each kind of nodes.

3.4.4 *Hardware Acceleration: an Illustrative Example*

While we will show real examples of such situation in Section 3.5.5, in this section we develop a theoretical model to illustrate how different jobs exhibiting different acceleration properties pose different

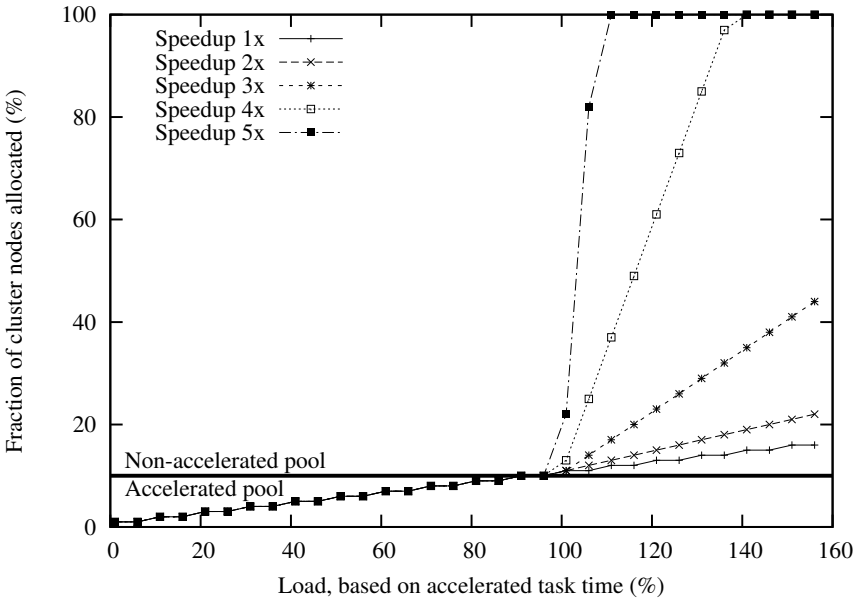


Figure 3.1: Slot allocation as a function of load

challenges to the scheduler. We consider a hypothetical cluster of machines running one accelerable job, and suppose that a fraction of the available nodes (10%) are enabled with hardware accelerators (the accelerated pool) while the remaining are not enabled with this technology (the non-accelerated pool).

As defined above, let $s^m_{req,acc}$ be the number of slots to be allocated in parallel to an accelerable job m in the accelerated pool to make its goal, TT the set of TaskTrackers in the cluster, and s_t the number of execution slots offered by a TaskTracker t. Then, we define the load of the system at any moment as:

$$load = \frac{s^m_{req,acc}}{\sum_{t \in P_{acc}} s_t} \tag{4}$$

Therefore, a load of 50% means that job m requires an allocation equivalent to the 50% of the slots available in the accelerated pool P_{acc} to meet its goal. Notice that this definition of load is based on $s^m_{req,acc}$ and thus, on μ^m_{acc}. Therefore, $load$ is calculated based on the average value μ^m_{acc} observed for the group of TaskTrackers of P_{acc}.

Figure 3.1 represents the effect of running accelerated tasks on the non-accelerated pool. Such situation is required when the load of the accelerated pool, as defined in (4), is beyond its capacity. This point can be seen in the figure when the allocated fraction of the cluster is above the size of the accelerated pool, indicated with the horizontal line and corresponding to 10% of the node count in the cluster. Beyond that point, accelerable tasks start running also on the non-accelerated pool, using non-accelerated versions of the code. Therefore, their performance is lower than when they run in the accelerated pool, and the difference depends on the per-task speedup of each job. In the figure we have included the simulation for different

jobs, each one with a different per-task speedup (from 1x to 5x). This example illustrates how jobs that show high per-tasks speedups in the accelerated pool will force the scheduler to steal resources from the non-accelerated pool to satisfy the demand of the job, missing the goal in many situations.

Section 3.5.5 will show this effect on real workloads running in a prototype on top of a medium-size cluster.

Finally, notice that while the presented mechanism assumes that only two pools are defined in the system, regular and accelerated, it could be easily extended to support different types of accelerators. Then, affinity could be enforced across pools based on the speedup observed for each one of them, being the generic pool the last on to use for accelerable jobs.

3.4.5 *Regarding Mappers and Reducers*

The discussion in the previous sections applies to both map and reduce tasks. Each TaskTracker has a number of slots for map tasks and number of slots for reduce tasks. MapReduce application are usually dominated by the time required to complete the map tasks, but cases where the reduce tasks dominate can also occur. In both cases, jobs start with a Map phase, in which performance data is collected, and is followed by the Reduce phase.

The scheduler cannot make assumptions about the reduce phase before completing reduce tasks. When a job is submitted, a job completion timeframe is derived from the distance between the present time and the completion time goal associated to the job. Both map and reduce phase must complete within the completion timeframe, one after the other.

In our system, a user can use the configuration files of Hadoop to provide an estimate of the relative cost of a reduce task compared to that of a map task. If no estimate is provided, the scheduler assumes that the computational cost of a map task is the same as that of a reduce task. As the number of map and reduce tasks to complete for a job is known in advance (when the input is split), the scheduler can estimate the cost of the reduce phase once the map phase is started.

3.4.6 *Modeling Application Characteristics*

This section evaluates different kinds of applications and how its characteristics affect the scheduler. The efficiency of the scheduler depends on the ability to determine the time at which the map phase should be completed in order to allow the reduce phase enough time to complete afterwards, and the assumption that the granularity of tasks in the map phase is enough to significantly change completion time of jobs through dynamic resource allocation. A job will only be

able to meet its goal if both phases complete on time; we analyze each phase separately.

During the map phase the scheduler leverages the malleability of the map tasks to complete within its timeframe by allocating cluster resources dynamically. Figure 3.2 shows the adaptability of the scheduler for jobs with different map task lengths and deadlines when said tasks take longer than initially expected. In particular the Figure not only shows when is it possible to correct a wrong estimation, but also the amount of additional resources needed to complete it while still meeting its completion goal, also described by the following equation: $s(d,l) = (100 \div l)/(d \div l)$.

The amount of slots (s) needed to adapt and correct a wrong estimation depends on the distance to the deadline (d) and the length of map tasks relative to the deadline (l). It can be measured comparing the number of sequential task executions, or waves, when the distance to the deadline is 100% with the number of waves when the distance is smaller. Hence the slots factor represents the additional slots needed. As expected from the baseline scenario, when the deadline is 100%, any kind of job, independently of its map task length, is able to complete with 1.0X of the estimated slots.

As shown in Figure 3.2, there is room to adapt to inaccurate estimations during the map phase since small variations make up for most cases. For instance, it's easy to adjust the scheduler when the inaccuracy is small and the distance to deadline is still close to 100%. It is also expected that the longer the task length, the more difficult it is for the scheduler to react. For jobs with smaller task lengths, there must be a huge inaccuracy in the estimation (>80-90%) to make it difficult for the scheduler to react, and that's to be expected since one of the keys of the MapReduce paradigm is to split the computation into smaller chunks. On the other hand, in the extreme case of a job with a map length bigger than 50%, meaning the number of map tasks is significantly smaller than the number of available slots and all the tasks are executed in a single wave: there is little the scheduler can do, but simply because there is no way to improve the scheduling.

While the scheduler leverages the malleability of the map phase to complete it within its timeframe, as described in Section 3.3, the time committed to the reduce phase given a completion time goal is derived from user estimates, and any mistake in this estimation will result in errors. The gray gradient in Figure 3.3 shows the impact of wrong reduce estimations on the total execution time (assuming perfect malleability of the map phase), which can also be explained with the equation: $d(rl,rd) = (rl \times rd)/100$, where the deviation d is a function of the reduce deviation (rd), and the length of the reduce phase relative to the total length of the job (rl). In this case deviation of the job is then directly proportional to the deviation of the reduce phase and its weight within the job. As it can be observed

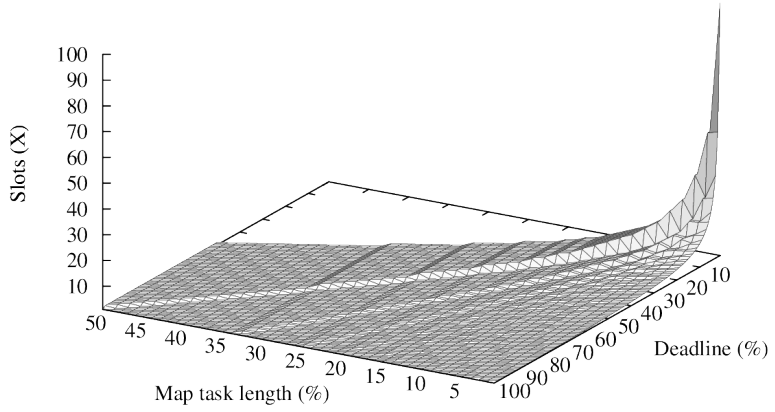


Figure 3.2: Slots needed depending on map task length

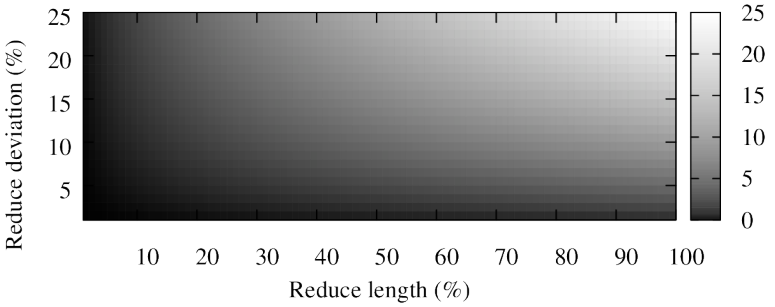


Figure 3.3: Distance to goal based on reduce length

in the Figure, which shows jobs with different reduce characteristics, the deviation caused by the reduce estimation is negligible when the reduce phase is small. For longer reduce phases, there must be a significant inaccuracy to affect the total execution; we have found in our experiments that inaccuracies in the estimations based on previous executions are always smaller than 5%.

3.5 EVALUATION

In this section we present four experiments to show the effectiveness of the proposed scheduling techniques. The experiments are based on the Adaptive Scheduler prototype for Hadoop which implements the task and job scheduling algorithms presented in previous sections.

3.5.1 *Workload*

The workload that we used for our experiments is composed of the following MapReduce applications:

- WordCount: takes a set of text files as input, and counts the number of times each word appears. This application has a regular task length distribution.
- Join: joins two data tables on a given column. Each map task processes a split of one of the two data tables—for each record

in the split, the mappers emit *key,value* where the key is the join key and the value is the record (tagged to indicate which of the two tables it came from). The reducers separate input records according to the tag, and perform a cross-product on the resulting two sets of records.

- Simulator: execution harness for a placement algorithm [18]. By varying the numbers of nodes and applications, in addition to the memory and CPU capacities of the nodes and demands of the applications, the algorithm can be made to execute for different lengths of time. Input data is negligible.
- Sort: sorting application as distributed in Hadoop. Both *map()* and *reduce()* are basically identity functions, and the main work of the application is performed by the internal runtime functions.
- Montecarlo: CPU intensive application with little input data. Two implementations of the *map()* function have been used: one written in pure Java, and another using Cell/BE-accelerated code.
- Crypt: represents data-intensive accelerable applications. We have two implementations of a 128-bit AES encryption algorithm: one written in Java, and another using Cell-accelerated code. Encrypts an input of 60GB in all experiments.

The specific workloads are described along with each experiment. Some applications, such as Simulator and Join, have an irregular distribution of task lengths. Also, both Montecarlo and Crypt benefit from executing on nodes with acceleration support (with a speedup of up to 25x and 2.5x respectively). Other applications do not benefit from acceleration.

3.5.2 *Execution Environment Description*

The four experiments can be grouped into two sets. The first set of experiments requires a homogeneous, general-purpose cluster to evaluate the efficiency of the completion time prediction technique using the Adaptive Scheduler with and without data affinity (experiments 1 and 2). For these experiments, we use a Hadoop cluster consisting of 61 2-way 2.1Ghz PPC970FX nodes with 4GB of RAM, connected using a gigabit ethernet.

In the second set of experiments (3 and 4) we evaluate the ability of the scheduler to dynamically manage heterogeneous pools of hardware. For these experiments we use a heterogeneous cluster, consisting of regular nodes and nodes enabled with acceleration support, to evaluate the scheduler with hardware affinity. The system used

to run this set of experiments is a 61 IBM QS22 cluster: each blade is equipped with 2x 3.2Ghz Cell processors and 8GB of RAM, and connected using gigabit ethernet. Due to technical restrictions in our facility we have not been able to integrate both clusters. Therefore, to simulate an environment in which only some of the nodes are enabled with accelerators, we create two logical partitions: a 54 node partition is considered to have no acceleration support (limited to the PPU of the Cell processors), and 6 nodes are accelerated (full access to SPUs).

Both sets of experiments were run using Hadoop 0.21. One of the nodes was configured to be both JobTracker and NameNode, and the 60 remaining nodes were used as DataNodes and TaskTrackers. Each TaskTracker was configured to run a maximum of one task in parallel (one slot for map tasks and one for reduce tasks).

3.5.3 *Experiment 1: Basic Adaptive Scheduler*

For this experiment we use a synthetic mix of applications to compose a realistic scenario of a MapReduce execution environment. The application set is composed of 4 different MapReduce applications that share resources during their execution: Simulator, Wordcount, Sort and Join. We configure each application with a particular completion time goal, derived from the completion time that each applications achieves when run in isolation. In this experiment two different scenarios have been evaluated. The first part of the experiment uses deadlines that all applications are able to meet, while the second part explores the behaviour of the scheduler when it's not possible to meet all the deadlines, and then compares the results with the Fair Scheduler.

In the first scenario, the set of applications is configured as follows: a big Simulator job (J1), which is configured to have a completion time goal of 6,000s (this is 1.69X its in-isolation completion time of 3,549s); a WordCount job (J2) configured with a completion time goal of 3,000s (2.37X its in-isolation completion time of 1,264s); a Sort job (J3) configured with a completion time of 3,000s (4.98X its in-isolation completion time of 602s); and two identical runs of the Join application (J4 and J5), each with a completion time goal of 150s (1.48X their in-isolation completion time of 101s each).

Figure 3.4 represents the execution of the workload over time. For the sake of clarity we group jobs by application into different rows. Each row contains solid and dotted lines, representing the number of running map and reduce tasks respectively. Jobs J1 to J5 are submitted at times S1 to S5, and the completion time goals are D1 to D5.

Simulator (J1) is the first job submitted and every slot is allocated to it, as there is nothing else in the system. When J2 is submitted, it shares resources with J1, and together they use every slot in the

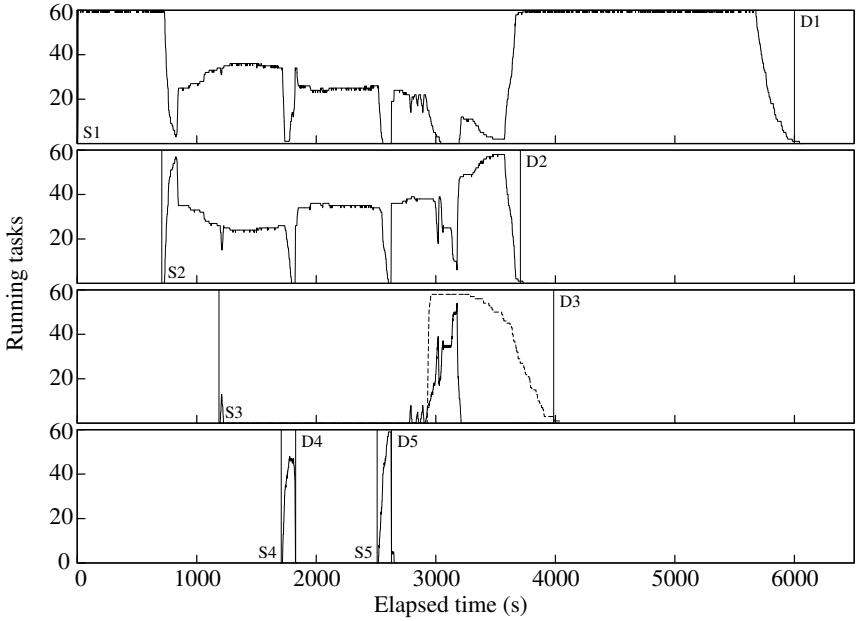


Figure 3.4: Adaptive Scheduler (solid: maps, dotted: reduces)

system. The scheduler allocates some slots to Sort (J3) when it is submitted, but as soon as the first map tasks are completed the slots return to J1 and J2 since they are estimated to have higher need of slots. Later, a short and high priority Join (J4) is submitted, and is allocated most of the resources in order to meet its goal. When J4 is finished, the resources return to J1 and J2. J3 resumes the execution of map tasks as it gets closer to its goal, leaving enough time to complete the reduce tasks. A second instance of Join (J5) is submitted, and again is assigned most of the resources. J2 completes close to its goal, at which point J1 is once again allocated the entire system and meets its goal.

In the second configuration of this experiment the completion time goals are set to be much tighter in order to show how the scheduler behaves when it's not possible to meet the completion time goals. The set of applications is: a Simulator (J1) with the same completion time goal of 6,000s (1.69X); Wordcount (J2) is now configured with a completion goal of 1,500s (1.18X); Sort (J3) is configured to complete after 650s (1.08X); and the Join executions (J4 and J5) are both configured with a completion time goal of 120s (1.18X).

As it can be observed in Figure 3.5, it is no longer possible to meet the goals of any of the jobs in the system since jobs J2-J5 have tighter completion time goals. But note that even though jobs are not meeting their completion time goals, slots are still being evenly distributed according to their relative size. On the other hand, J1 takes longer than in the previous configuration, but that is simply a side effect of the lack of resource awareness in Hadoop, which leads to variable results depending on how applications are mixed during its execution.

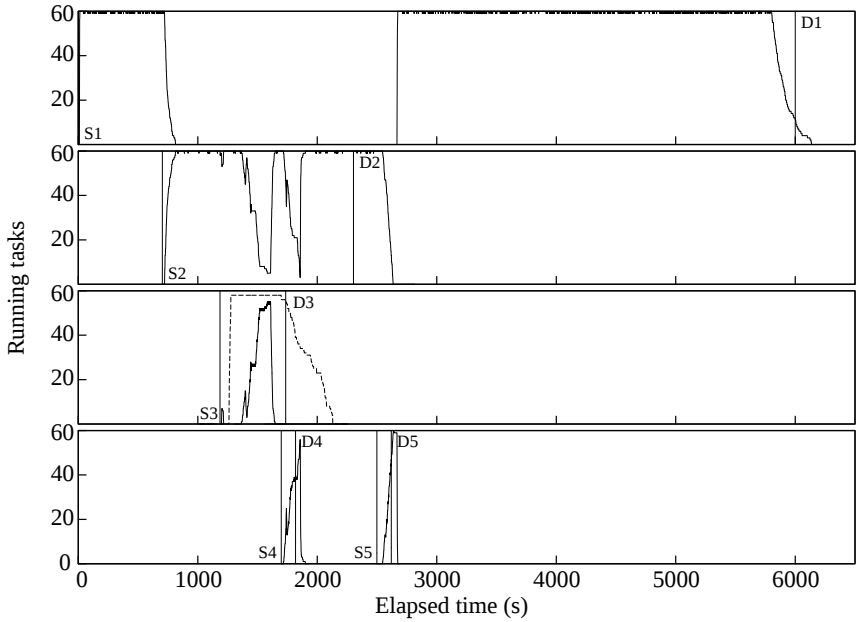


Figure 3.5: Adaptive Scheduler with tighter completion time goals (solid: maps, dotted: reduces)

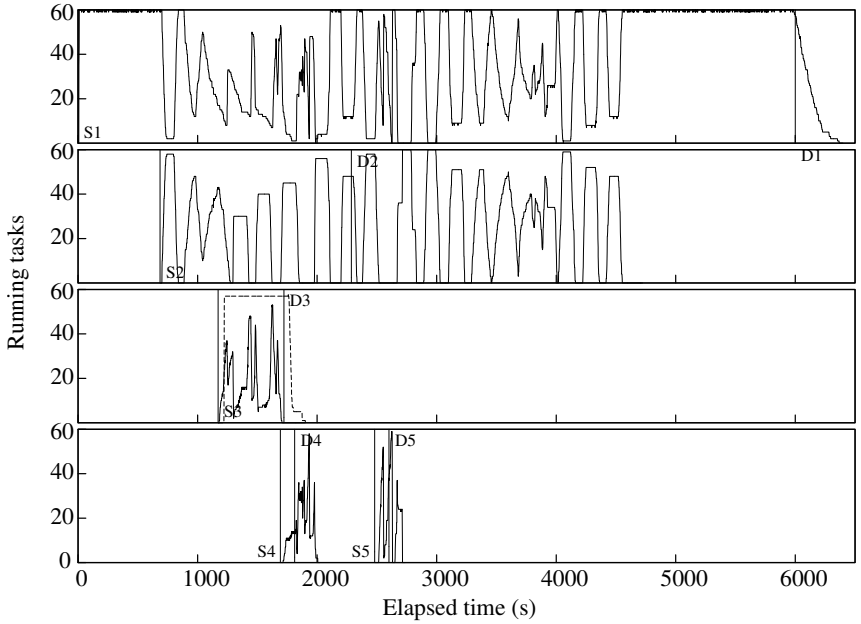


Figure 3.6: Fair Scheduler (solid: maps, dotted: reduces)

Once we had seen the effectiveness of the basic scheduler, we wanted to compare its behavior with a state of the art MapReduce scheduler. For such purpose, we used the Fair Scheduler [90], which uses job priorities to make scheduling decisions, in place of our completion time goal oriented scheduler. Figure 3.6 shows the execution of the Fair Scheduler with its default configuration. As expected, high-priority jobs miss their goal. This is especially noticeable for J2, J4 and J5, which take twice as much time as their desired completion time goals. J3 on the other hand does not take so long because it has a significant reduce phase, and there is not as much competition for reduce slots as there is for maps. Fair Scheduler supports weights to prioritize applications, which could be used to emulate the behaviour of completion time goals. However, weights are a static setting that need to be configured in advance, and could be challenging as the complexity of the workload increases. See [60] for an extended comparison with the Fair Scheduler considering different weights.

3.5.4 *Experiment 2: Scheduling with Data Affinity*

Experiment 2 illustrates how the scheduler simultaneously manages two different goals when running a mixed set of jobs concurrently: meeting completion time goals and maximizing data locality. Notice that the former is the main scheduling criteria, while the latter is a best effort approach.

For this purpose we ran two different tests: with and without completion time goals. We evaluate two different configurations of the Adaptive Scheduler with Data Affinity: setting the maximum delay per remote task to either one or three attempts. Experiments are executed twice: once with the block replication factor set to one, and another one with replication set to three.

3.5.4.1 *Workload without completion time goals*

This part aims to determine if the Adaptive Scheduler enhanced with data affinity considerations is able to improve the percentage of local tasks for a set of applications that have relaxed completion time goals. The relaxed completion time goals mean that the scheduler can concentrate on achieving data locality. We compare the results achieved by the Adaptive Scheduler with Data Affinity (maximum delay set to one and to three) with both the Basic Adaptive Scheduler and the Fair Scheduler.

This experiment focuses on the locality of map tasks, and so the workload is composed of two instances of the Join application and three instances of WordCount. The Simulator application is not used since it has a negligible amount of data, so the number of local task will not vary regardless of the scheduler used, and Sort is not used because it is mostly reduce-oriented. We used a balanced distribution

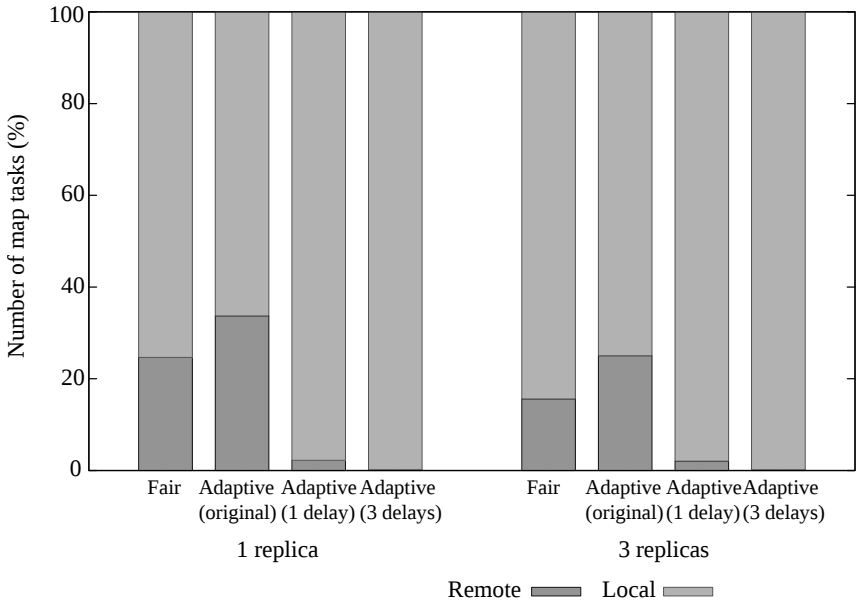


Figure 3.7: Data locality without completion time

of the input data of the applications across all the nodes storing data. This data distribution permits the Fair Scheduler to prioritize the execution of local tasks and thus to achieve a high percentage of locality. This is because each node stores the same amount of data and each map task of these applications is fed with the same amount of bytes (one data block).

Figure 3.7 shows the percentage of local tasks achieved for the evaluated configurations. We can see that all configurations achieve higher locality percentage when using three replicas per block than when using just one. For both replication factors, the configuration that exhibits the highest percentage of remote tasks is the Basic Adaptive Scheduler, followed by the Fair Scheduler with 24.6% of tasks executed remotely with one replica per block, and 15.6% executed remotely with three replicas. Configurations that use the Adaptive Scheduler with Data Affinity significantly improve on these percentages. In the worst case, when just one delay is allowed per remote task and the replica factor is one, the percentage of local tasks achieved is around 98%, and in the best scenario, allowing three delays per remote tasks and using three replicas per block, almost all tasks (99.8%) are executed local to their data.

Table 3.1 shows the benefits that this improvement in data locality may have on the execution of the applications. As it can be seen in the amount of data transmitted across the network, when there are three replicas per block, the Adaptive Scheduler with Data Affinity uses 82.6% less network bandwidth than that used by the Fair Scheduler. If the replication factor is one, then the reduction in the required network bandwidth is around 70.7%. Notice that although the percentage of local tasks achieved by the Fair Scheduler is greater than

SCHEDULER	BLOCK REPLICAS	DATA VOLUME
Fair Scheduler	1	40.94 GB
	3	33.48 GB
Basic Adaptive Scheduler	1	21.24 GB
	3	10.47 GB
Adaptive Scheduler with data-affinity (1 delay)	1	12.00 GB
	3	6.16 GB
Adaptive Scheduler with data-affinity (3 delays)	1	12.00 GB
	3	5.83 GB

Table 3.1: Network Bandwidth: non-restricted completion time goal

the percentage achieved by the Basic Adaptive Scheduler, the amount of data transmitted across the network with the Fair Scheduler is bigger. This is because the input data required by the tasks that happen to be remote in the case of the Fair Scheduler is greater than the input data required by the remote tasks in the case of the Basic Adaptive Scheduler.

Despite the improvements in network bandwidth usage, the overall makespan of the workload is not significantly different in this execution environment. For instance, a map task of the Wordcount application takes an average of 92±3s when executed locally, and 93±4s when executed remotely. Both times are almost the same since Wordcount needs more CPU and input bandwidth is not critical. However, higher performance improvement is expected under other environments, with a different network topology, or with other applications that are more sensitive to the performance of reading the input. To check the latter assumption we measured the map length of the Sort (which is an identity function and thus its time is bounded by the time of pulling the input data) and we observed that it takes 12±2s for local tasks and 17±3s for remote tasks.

3.5.4.2 *Workload with completion time goals*

This test evaluates the efficiency of the Adaptive Scheduler with Data Affinity when the applications have tight deadlines. In this situation, meeting the performance goal is the main goal of the Adaptive Scheduler and thus, remote task will be delayed only if this decision does not compromise the performance goals. The purpose of the experiment is twofold: on one hand we want to measure the percentage of task locality achieved when this is not the main criteria; on the other hand we want to evaluate if deferring some remote tasks makes

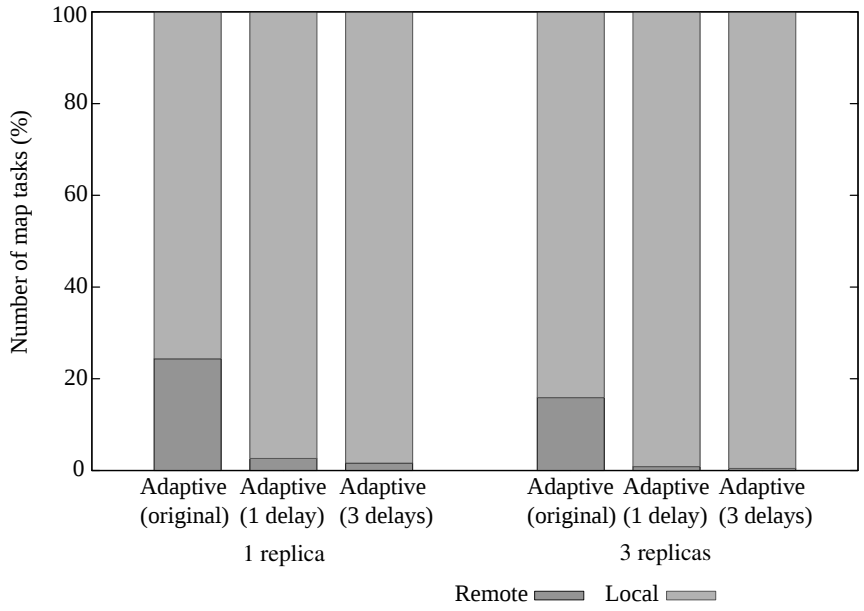


Figure 3.8: Data locality with completion time

the scheduler fail in making the applications meet their performance goals.

The workload used for these executions is the same as the one described in section 3.5.4.1: three instances of WordCount and two instances of Join. We executed this workload using both Basic Adaptive Scheduler and with Data Affinity.

Figure 3.8 shows the percentage of local tasks achieved for all the executions of the workload. The percentage of local tasks increases for all the configurations if we increase the number of replicas per block from one to three. The Adaptive Scheduler with Data Affinity always achieves a higher percentage of local tasks. This happens even if we compare the most restrictive configuration with data affinity (one replica per block and just one delay per remote task) and the least restrictive configuration without data affinity (three replicas and a limit of three delays per remote tasks). Observe that in the least restrictive configuration, the percentage of local tasks achieved with Data Affinity is close to 100%.

This percentage of local tasks is achieved without compromising the ability of the jobs to achieve their completion time goals. This can be seen looking at the execution over time of the workload. Figure 3.9 and Figure 3.10 show the results when the maximum delay per remote task is set to three and the block replication factor is set to one and to three respectively. WordCount jobs are submitted at time S_1, S_2 and S_3 respectively and their completion time goals are D_1, D_2 and D_3; Join jobs are submitted at S_4 and S_5, and their completion time goals are D_4 and D_5. We can see in the graphs that all WordCount jobs meet their completion time goal. The execution with 3 replicas shows a better performance than the execution with a single replica

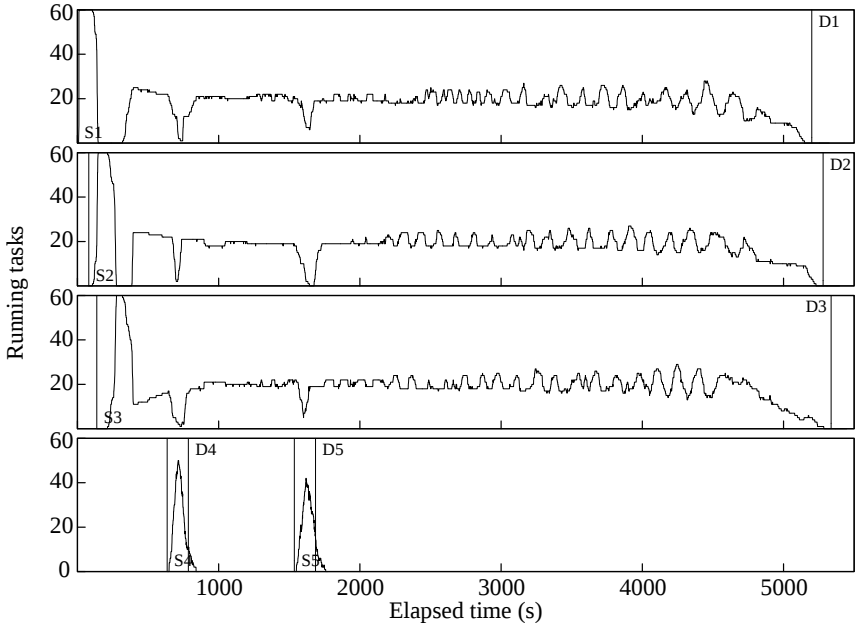


Figure 3.9: Adaptive with data-affinity (3 delays, 1 replica)

for all of them, as the specified delay to improve data locality is lower. However, with one replica, the performance achieved by the jobs is still affordable as it is lower than their performance goal. Join jobs, with 3 replicas, also meet their completion time goal. However, when the replication factor is one (and thus more tasks are remote), both instances of Join miss their completion time goal. The characteristics of this job makes it more sensitive to misestimations and possible delays of remote tasks: they are short jobs (with tight completion time goals and few tasks to schedule) and so there are few chances to correct the effects of a wrong decision. Also, this configuration – with just one replica – is not the usual scenario for MapReduce applications as it limits the reliability. In these graphs we can see that the Join jobs are able to get more concurrent slots than when setting to three the maximum delay per remote task and thus, to improve their performance. In the case of the execution with one replica per block, one of the Join instances makes its performance goal and the other misses it just slightly. Recall that the percentage of local task for this configuration is around 98% (see figure 3.8) and thus, although this is not the configuration that gets the highest data locality percentage, it is a reasonable configuration candidate if we need to set the replication factor to one and if we have to execute applications with few tasks and very tight completion time goals.

3.5.5 *Experiment 3: Scheduling with Hardware Affinity*

In this experiment we evaluate the execution of the same workload using two different configurations, and show the benefits of adding

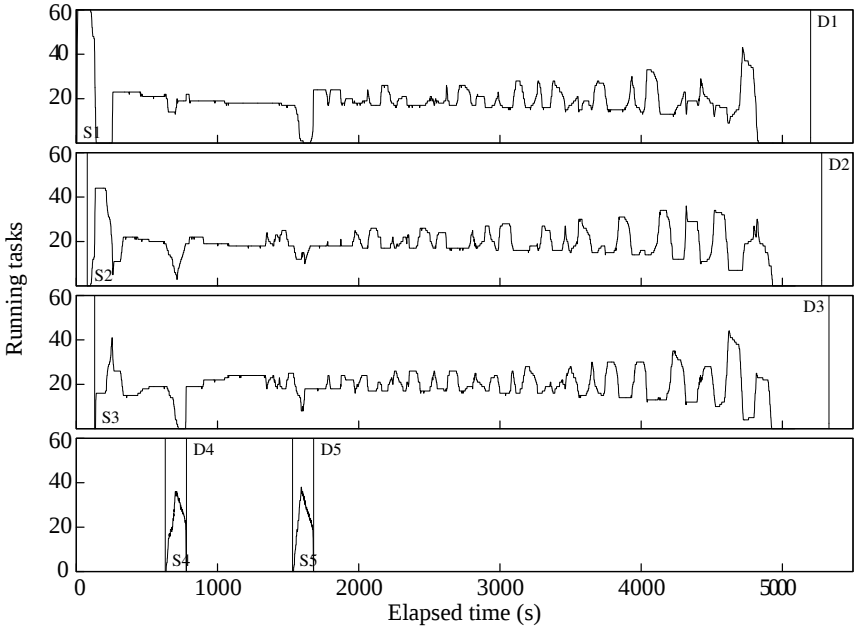


Figure 3.10: Adaptive with data-affinity (3 delays, 3 replicas)

hardware affinity support to the Adaptive Scheduler when running simultaneously accelerable and non-accelerable applications. The configurations are as follows:

- Configuration 1.1. Adaptive Scheduler with Hardware Affinity and 10% of the nodes enabled with hardware acceleration. In this case, the scheduler prioritizes allocation of accelerator-enabled nodes to accelerable applications, as described in 3.4.3.
- Configuration 1.2. Basic Adaptive Scheduler. Although this scheduler does not distinguish the kind of hardware, 10% of the nodes are still enabled with hardware acceleration support. Thus, all map tasks assigned to accelerated nodes execute the accelerated code (if available).

The workload is composed of one instance of the WordCount application, which is not able to exploit hardware acceleration support, and two instances of the Montecarlo simulation, which exhibit a high speedup on accelerated nodes. The first job that is submitted is the WordCount application. Afterwards, the first Montecarlo simulation is submitted at S2, and a second Montecarlo job is submitted 300s *after* the first one completes, at time S3. Recall that S3 will vary depending on the actual completion time of the first instance. WordCount is set to have a relaxed completion time goal –5,000s (D1)–, while Montecarlo jobs have tighter completion time goals – 2,200s (D2) for the first instance and 500s (D3) for the second one. The goal for the first instance of Montecarlo can be met running only with resources from the accelerated pool, while the goal for the second one is so tight that

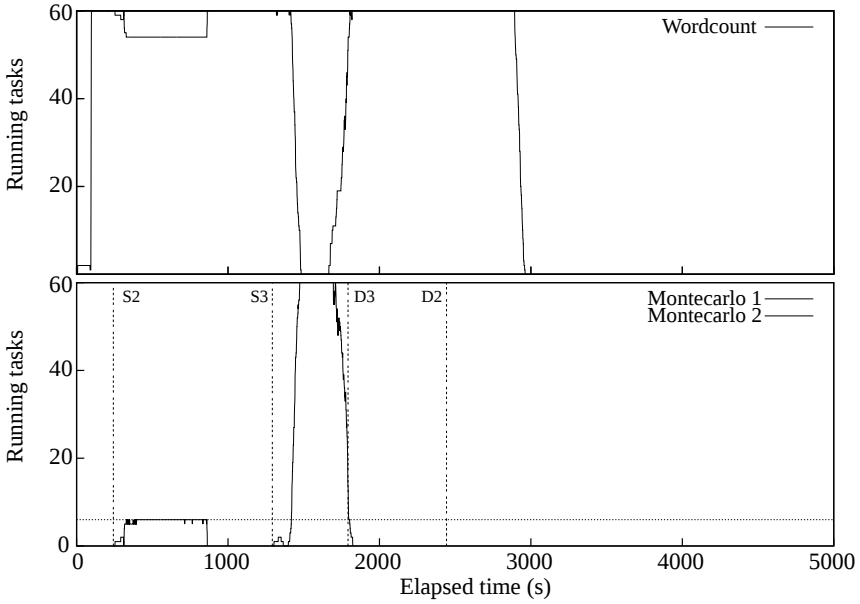


Figure 3.11: Adaptive with Hardware Affinity

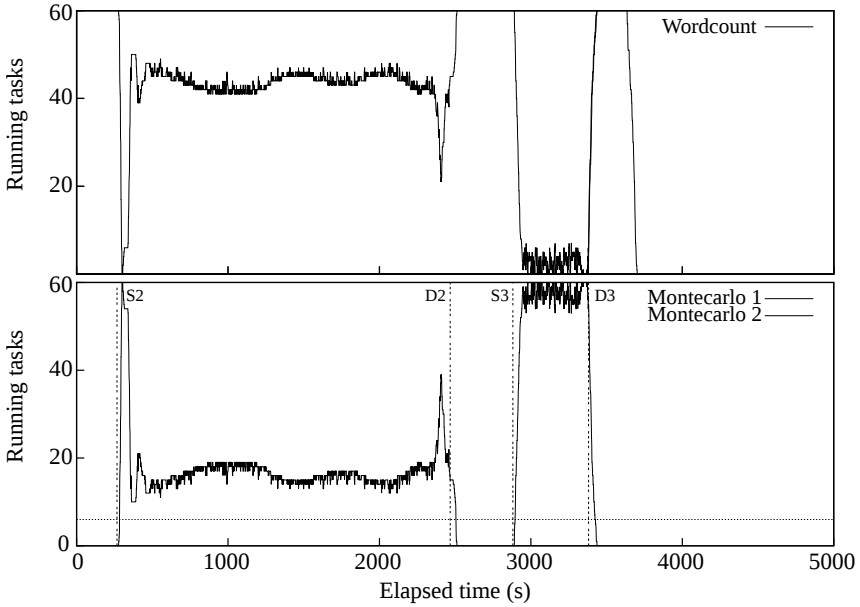


Figure 3.12: Basic Adaptive Scheduler

both accelerated and non-accelerated resources must be used to meet it.

Figure 3.11 shows the results for Configuration 1.1. An horizontal line marks the limit of the accelerated pool. Recall that WordCount initially runs across the two pools, but without exploiting the acceleration capabilities of the accelerated pool, because no accelerated code is provided for this application. When the first Montecarlo job is submitted (S2), the Adaptive Scheduler starts the calibration phase: executing two tasks for this job, one in each pool, to evaluate the speedup. When the first estimation of resource demand is done, the scheduler starts allocating resources to the job in the accelerated pool. The job completes at time 863s. Shortly after that, the second Montecarlo instance is submitted (S3). Due to its tighter completion time goal, and after the initial execution of one task in each partition to calibrate the estimation, the scheduler determines that for this second instance to meet its goal, the job needs to be spread across both partitions. The job meets its deadline and completes at 1824s. Note that this second Montecarlo instance performs slightly faster than the first Montecarlo instance, although it is running most of the time using 9x times more nodes than the first one (this situation is deeply analysed in Section 3.4.3). After that, WordCount gets all the resources again until completion.

Figure 3.12 shows the results for Configuration 1.2, in which the Adaptive Scheduler is not aware of hardware heterogeneity. In this case, all maps of all jobs execute across all nodes. As shown, the number of nodes assigned to the first Montecarlo job is higher than when using the Adaptive Scheduler with hardware affinity. The reason for this is that, as the Adaptive Scheduler assigns nodes to this job that do not have hardware acceleration support, the execution of the non-accelerated version of these maps increases considerably the average map time for this job (recall that this non-accelerated version is around 25x slower than the accelerated version). Thus the job requires more nodes to meet its completion time goal. However, in spite of the higher number of assigned nodes, the execution time of this job is still more than twice the execution of the same job under the same execution conditions but using hardware affinity during the scheduling. Note also that this configuration also increases the execution time of the WordCount application. This is because, as this application has a very relaxed completion time goal, it has a low priority for the scheduler that assigns to WordCount only the nodes that the Montecarlo simulation does not need to meet its tight goal. Regarding the execution of the second job of the Montecarlo simulation there are not noticeable differences between this configuration and the configuration considering hardware affinity because its tight completion time goal requires in both configurations to get most of the nodes in the cluster.

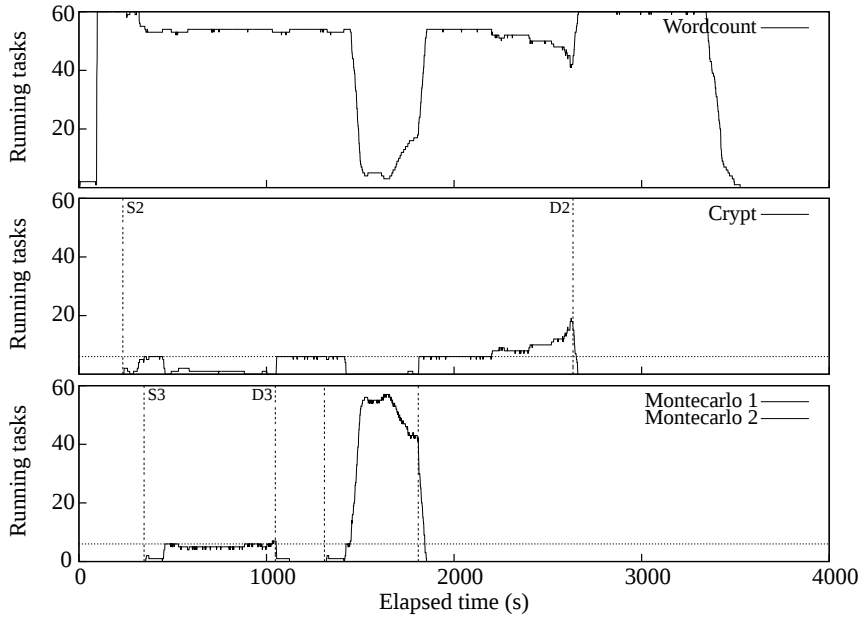


Figure 3.13: Heavy load on accelerated pool

3.5.6 *Experiment 4: Arbitrating Between Pools*

In this experiment we evaluate the Adaptive Scheduler when arbitration between both pools is required. We use a workload composed of a WordCount job (which is not accelerable), the Montecarlo simulation (which is accelerable and has a high per-task speedup) and the Crypt application (which is accelerable and has a moderate per-task speedup).

We show the results for the following two configurations:

- Configuration 2.1. Shows how the Adaptive Scheduler arbitrates allocation inside the accelerated pool when accelerable jobs are competing. In addition, the tight deadline for accelerable jobs forces the scheduler to allocate them nodes from the non-accelerated pool.
- Configuration 2.2. Illustrates how a non-accelerable job can steal nodes from the accelerated pool when needed to meet its completion time goal.

Figure 3.13 shows the result for Configuration 2.1, in which the load on the accelerated pool is high. We execute one instance of Crypt and Wordcount and two instances of Montecarlo, in order to increase the load on the accelerated pool. A horizontal line in the graphs marks the number of nodes in the accelerated pool.

We first launch the job that executes the WordCount application. After determining that it is not accelerable, the scheduler assigns to the job all the available resources in the cluster until the job that executes the Crypt application is submitted (S2). Once the scheduler

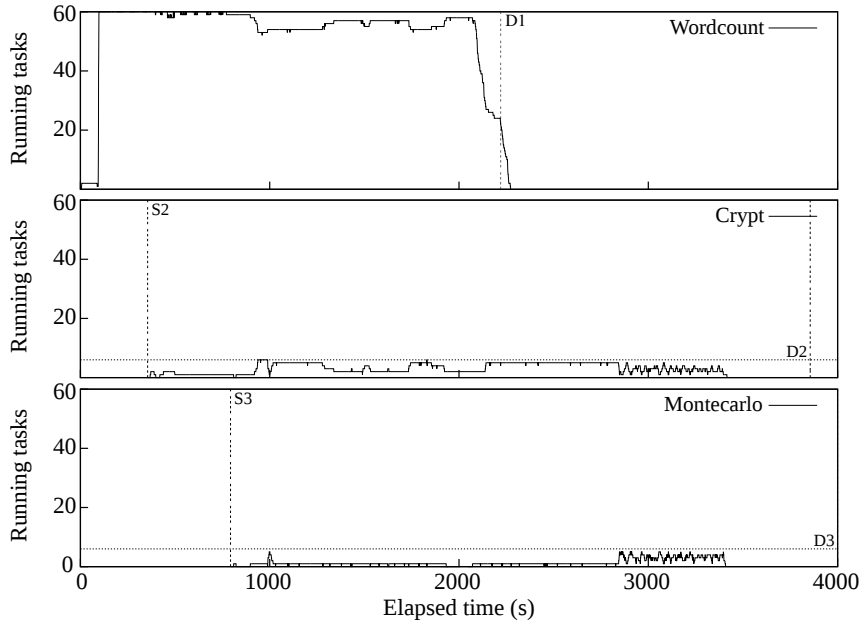


Figure 3.14: Heavy load on non-accelerated pool

decides that this job is accelerable, it starts applying the affinity criteria: as WordCount map tasks running on accelerated nodes finish, the scheduler assigns those nodes to Crypt. When the job for the Montecarlo simulation starts (S3), both jobs have to share the accelerated pool. The scheduler decides how to allocate these nodes considering the completion time goal of each job. In this example, Montecarlo has a tighter completion time than Crypt and therefore gets more nodes than Crypt. At the same time, WordCount continues executing on nodes from the non-accelerated pool, as they are enough for this job to meet its completion time goal. After the first instance of the Montecarlo simulation is completed, a second instance of Montecarlo is submitted, in this case with a very tight completion time goal. This forces the Adaptive Scheduler to assign most of the nodes in the cluster (from both the accelerated and non-accelerated pools) to this job. When this job completes, Crypt and WordCount continue to run using only the nodes from the accelerated and non-accelerated pool respectively, until the scheduler detects that Crypt does not have enough resources to meet its completion time goal. At this point, the scheduler starts allocating some non-accelerated nodes to Crypt as well. When Crypt completes, WordCound starts running across all nodes in the cluster and finally meets its goal. Notice from this experiment that when the scheduler detects that accelerated nodes are not enough to meet the completion time goal of accelerable job (Crypt and Montecarlo), the number of non-accelerated nodes required to compensate the shortage of accelerated nodes is considerably higher for Montecarlo than for Crypt. As discussed in Section 3.4.3, this is

due to the different per-task speedup of both jobs: 25x in the case of Montecarlo tasks and 2.5x in the case of Crypt tasks.

Figure 3.14 shows the results for Configuration 2.2, in which the accelerable jobs have relaxed completion time goals and the non-accelerable job is submitted with a very tight completion goal. We execute one instance of each application. Initially, the scheduler estimates that WordCount and Crypt will be able to meet their goals. But shortly after Montecarlo is submitted, the scheduler notices that WordCount will need more resources to meet its goal and thus claims some nodes from the accelerated pool. After WordCount completes, the remaining jobs continue sharing the accelerated pool, but most of it is assigned to Crypt since it has a higher need of slots. Once the scheduler acknowledges that Crypt too will meet its completion goal and Montecarlo becomes more needy (at around time 2,850s), both jobs start sharing the pool more equally until completion.

3.6 RELATED WORK

Process scheduling is a deeply explored topic for parallel applications, considering different type of applications, different scheduling goals and different platform architectures ([30]). There has also been some work focused on adaptive scalable schedulers based on job sizes ([43, 84]), but in addition to some of these ideas, our proposed scheduler takes advantage of one of the key features of MapReduce: the fact that jobs are composed of a large number of similar tasks.

MapReduce scheduling has been discussed in the literature, and different approaches have been presented. The initial scheduler provided by the Hadoop distribution uses a very simple FIFO policy, considering five different application priorities. In addition, in order to isolate the performance of different jobs, the Hadoop project is working on a system for provisioning dedicated Hadoop clusters to applications [7], but this approach can result in resource underutilization. There are several proposals of fair scheduling implementations to manage data-intensive and interactive applications executed on very large clusters for MapReduce environments ([90, 91]) and for Dryad ([41, 42]). The main concern of these scheduling policies is to give equal shares to each user and achieve maximum utilization of the resources. However, scheduling decisions are not dynamically adapted based on job progress, so this approach is not appropriate for applications with different performance goals.

There have been other proposals that involve setting high-level completion goals for MapReduce applications. In addition to our initial implementation [60], others have shown interest in this particular topic. FLEX [86] is a scheduler proposed as an add-on to the Fair Scheduler to provide Service-Level-Agreement (SLA) guarantees. More recently [82] introduces a novel resource management frame-

work that consists of a job profiler, a model for MapReduce jobs and a SLO-scheduler based on the Earliest Deadline First scheduling strategy.

In [70], the authors introduce a system to manage and dynamically assign the resources of a shared cluster to multiple Hadoop instances. Priorities are defined by users using high-level policies such as budgets. This system is designed for virtualized environments, unlike the proposed work, which is implemented as a regular Hadoop MapReduce scheduler and thus is able to run on standard Hadoop installations and provide more accurate estimations.

Regarding the execution of MapReduce applications on heterogeneous hardware, in [89] the authors consider the influence that hardware heterogeneity may have on the scheduling of speculative tasks. Our proposal in orthogonal to this one as we do not face the scheduling of speculative tasks and we have not enable this option in the configuration of our execution environment. In [5] the authors focus on avoiding stragglers (which may cause the execution of speculative tasks). They show that most of them are due to network traffic. Thus, although dealing with stragglers is not the focus of our proposal, our scheduler is also avoiding them as the percentage of local task that it is able to achieve is around 100%. There are several works in the literature that consider the heterogeneity trend on current execution platforms. [72] studies the impact of heterogeneity on large clusters and presents techniques to include task placement constraints.

More recently, Hadoop schedulers have focused on being more aware of both resources available in each node and resources required by applications [11]. In [63] we adapt the Adaptive Scheduler to be resource-aware.

3.7 SUMMARY

This chapter presents a scheduler for multi-job MapReduce environments that is able to dynamically build performance models of the executing workloads, and then use these models for scheduling purposes. This ability is leveraged to adaptively manage workload performance while observing and taking advantage of the particulars of the execution environment of modern data analytics applications, such as hardware heterogeneity and distributed storage.

The scheduler targets a highly dynamic environment in which new jobs can be submitted at any time with different user-defined completion time goals. Thus the actual amount of resources available for applications can vary over time depending on the workload. Beyond the formulation of the problem and the description of the scheduling algorithm and technique, a working prototype called Adaptive Scheduler has been implemented. Using the prototype and medium-sized clusters (of the order of tens of nodes), the following aspects

have been studied separately: the scheduler's ability to meet high-level performance goals guided only by user-defined completion time goals; the scheduler's ability to favor data-locality in the scheduling algorithm; and the scheduler's ability to deal with hardware heterogeneity, which introduces hardware affinity and relative performance characterization for those applications that can benefit from executing on specialized processors.

The work described in this chapter is a summary of the following main publications:

[62] Jordà Polo, David Carrera, Yolanda Becerra, Jordi Torres, Eduard Ayguadé, Malgorzata Steinder, and Ian Whalley. Performance-driven task co-scheduling for MapReduce environments. In *Network Operations and Management Symposium, NOMS*, pages 373–380, Osaka, Japan, 2010

[61] Jordà Polo, David Carrera, Yolanda Becerra, Jordi Torres, and Eduard Ayguadé. Performance Management of Accelerated MapReduce Workloads in Heterogeneous Clusters. In *ICPP '10: Proceedings of the 39th IEEE/IFIP International Conference on Parallel Processing*, San Diego, CA, USA, 2010

[64] Jordà Polo, Yolanda Becerra, David Carrera, Malgorzata Steinder, Ian Whalley, Jordi Torres, and Eduard Ayguadé. Deadline-Based MapReduce Workload Management. *IEEE Transactions on Network and Service Management*, pages 1–14, 2013-01-08 2013. ISSN 1932-4537

4

SCHEDULING WITH SPACE AND TIME CONSTRAINTS

4.1 INTRODUCTION

In recent years, the industry and research community have witnessed an extraordinary growth in research and development of data-analytic technologies, and the adoption of MapReduce [24] has been pivotal to this phenomenon. Pioneer implementations of MapReduce [8] have been designed to provide overall system goals (e.g. job throughput). Thus, support for user-specified goals and resource utilization management have been left as secondary considerations at best. But both capabilities are arguably crucial for the further development and adoption of large-scale data processing. On one hand, more users wish for ad-hoc processing in order to perform short-term tasks [79]. Furthermore, in a cloud environments users pay for resources used. Therefore, providing consistency between price and the quality of service obtained is key to the business model of the cloud. Resource management, on the other hand, is also important as providers are motivated by profit and hence require both high levels of automation and resource utilization while avoiding bottlenecks.

The main challenge in enabling resource management in Hadoop clusters stems from the resource model adopted in MapReduce. Hadoop expresses capacity as a function of the number of tasks that can run concurrently in the system. To enable this model the concept of *slot* was introduced as the minimum schedulable unit in the system. Slots are bound to a particular type of task, either reduce or map, and one task of the appropriate type is executed in each slot. The main drawback of this approach is that slots are fungible across jobs: a task

(of the appropriate type) can execute in any slot, regardless of the job of which that task forms a part.

This loose coupling between scheduling and resource management limits the opportunity to efficiently control the utilization of resources in the system. Providing support for user-specified goals in MapReduce clusters is also challenging, due to high variability induced by the presence of outlier tasks (tasks that take much longer than other tasks) [5, 59, 86, 82]. Solutions to mitigate the detrimental impact of such outliers typically rely on scheduling techniques such as speculative scheduling [89], and killing and restarting of tasks [5]. These approaches, however, may result in wasted resources and reduced throughput. More importantly, all existing techniques are based on the typed-slot model and therefore suffer from the aforementioned limitations.

This chapter presents a Resource-aware Adaptive Scheduler for MapReduce [1] (hereafter RAS), capable of improving resource utilization and which is guided by completion time goals. In addition, it also addresses the system administration issue of configuring the number of slots for each machine, which—as will be demonstrated—has no single, homogeneous, and static solution on a multi-job MapReduce cluster.

While existing work focuses on the typed-slot model—wherein the number of tasks per worker is fixed throughout the lifetime of the cluster, and slots can host tasks from any job—the proposed approach offers a novel resource-aware scheduling technique which advances the state of the art in several ways:

- Extends the abstraction of 'task slot' to 'job slot'. A 'job slot' is job specific, and has an associated resource demand profile for map and reduce tasks.
- Leverages resource profiling information to obtain better utilization of resources and improve application performance.
- Adapts to changes in resource demand by dynamically allocating resources to jobs.
- Seeks to meet soft-deadlines via a utility-based approach.
- Differentiates between map and reduce tasks when making resource-aware scheduling decisions.

The structure of this chapter is as follows. The scheduler's design and implementation is described in detail in Section 4.2. An evaluation of our prototype in a real cluster is is presented in Section 4.3. And finally, Section 4.4 discusses the related work.

4.2 RESOURCE-AWARE ADAPTIVE SCHEDULER

The driving principles of RAS are resource awareness and continuous job performance management. The former is used to decide task placement on TaskTrackers over time, and is the main object of study of this chapter. The latter is used to estimate the number of tasks to be run in parallel for each job in order to meet some performance objectives, expressed in the form of completion time goals, and as described in Chapter 3 was extensively evaluated and validated in [59].

In order to enable this resource awareness, this proposal introduces the concept of 'job slot'. A job slot is an execution slot that is bound to a particular job, and a particular task type (reduce or map) within that job. This is in contrast to the traditional approach, wherein a slot is bound only to a task type regardless of the job. The rest of the chapter will use the terms 'job slot' and 'slot' interchangeably. This extension allows for a finer-grained resource model for MapReduce jobs. Additionally, the scheduler determines the number of job slots, and their placement in the cluster, dynamically at run-time. This contrasts sharply with the traditional approach of requiring the system administrator to statically and homogeneously configure the slot count and type on a cluster. This eases the configuration burden and improves the behavior of MapReduce clusters.

Completion time goals are provided by users at job submission time. These goals are treated as soft deadlines in as opposed to the strict deadlines familiar in real-time environments: they simply guide workload management.

4.2.1 *Problem Statement*

We are given a set of MapReduce jobs $\mathcal{J} = \{1, \ldots, J\}$, and a set of TaskTrackers $\mathcal{TT} = \{1, \ldots, TT\}$. We use j and tt to index into the sets of jobs and TaskTrackers, respectively. With each TaskTracker tt we associate a series of resources, $\mathcal{R} = \{1, \ldots, R\}$. Each resource of TaskTracker tt has an associated capacity $\Omega_{tt,1}, \ldots, \Omega_{tt,r}$. In our work we consider disk bandwidth, memory, and CPU capacities for each TaskTracker. Note that extending the algorithm to accommodate for other resources, e.g., storage capacity, is straightforward.

A MapReduce job (j) is composed of a set of tasks, already known at submission time, that can be divided into map tasks and reduce tasks. Each TaskTracker tt provides to the cluster a set of job-slots in which tasks can run. Each job-slot is specific for a particular job, and the scheduler will be responsible for deciding the number of job-slots to create on each TaskTracker for each job in the system.

Each job j can be associated with a completion time goal, T^j_{goal}, the time at which the job should be completed. When no completion time goal is provided, the assumption is that the job needs to be completed

at the earliest possible time. Additionally, with each job we associate a resource consumption profile. The resource usage profile for a job j consists of a set of average resource demands $\mathcal{D}_j = \{\Gamma_{j,1}, \ldots, \Gamma_{j,r}\}$. Each resource demand consists of a tuple of values. That is, there is one value associated for each task type and phase (map, reduce in shuffle phase, and reduce in reduce phase, including the final sort).

We use symbol P to denote a placement matrix of tasks on TaskTrackers, where cell $P_{j,tt}$ represents the number of tasks of job j placed on TaskTracker tt. For simplicity, we analogously define P^M and P^R, as the placement matrix of Map and Reduce tasks. Notice that $P = P^M + P^R$. Recall that each task running in a TaskTracker requires a corresponding slot to be created before the task execution begins, so hereafter we assume that placing a task in a TaskTracker implies the creation of an execution slot in that TaskTracker.

Based on the parameters described above, the goal of the scheduler presented in this chapter is to determine the best possible placement of tasks across the TaskTrackers as to maximize resource utilization in the cluster while observing the completion time goal for each job. To achieve this objective, the system will dynamically manage the number of job-slots each TaskTracker will provision for each job, and will control the execution of their tasks in each job-slot.

4.2.2 *Architecture*

Figure 4.1 illustrates the architecture and operation of the resource-aware scheduler. The system consists of five components: Placement Algorithm, Job Utility Calculator, Task Scheduler, Job Status Updater and Job Completion Time Estimator.

Most of the logic behind RAS resides in the JobTracker. We consider a scenario in which jobs are dynamically submitted by users. Each submission includes both the job's completion time goal (if one is provided) and its resource consumption profile. This information is provided via the job configuration XML file. The JobTracker maintains a list of active jobs and a list of TaskTrackers. For each active job it stores a descriptor that contains the information provided when the job was submitted, in addition to state information such as number of pending tasks. For each TaskTracker (TT) it stores that TaskTracker's resource capacity (Ω_{tt}).

For any job j in the system, let s^j_{pend} and r^j_{pend} be the number of map and reduce tasks pending execution, respectively. Upon completion of a task, the TaskTracker notifies the **Job Status Updater**, which triggers an update of s^j_{pend} and r^j_{pend} in the job descriptor. The Job Status Updater also keeps track of the average task length observed for every job in the system, which is later used to estimate the completion time for each job.

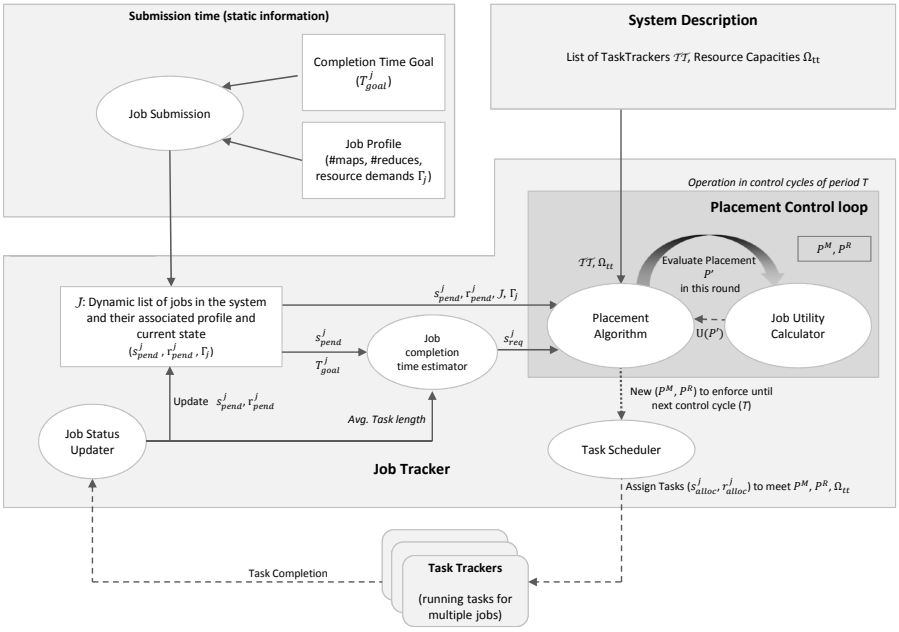


Figure 4.1: System architecture

The **Job Completion Time Estimator** estimates the number of *map* tasks that should be allocated concurrently (s^j_{req}) to meet the completion time goal of each job. To perform this calculation it relies on the completion time goal T^j_{goal}, the number of pending *map* tasks (s^j_{pend}), and the observed average task length. Notice that the scenario we focus on is very dynamic, with jobs entering and leaving the system unpredictably, so the goal of this component is to provide estimates of s^j_{req} that guide resource allocation. This component leverages the techniques already described in [59].

The core of RAS is the Placement Control loop, which is composed of the **Placement Algorithm** and the **Job Utility Calculator**. They operate in control cycles of period T, which is of the order of tens of seconds. The output of their operation is a new placement matrix P that will be active until the next control cycle is reached (current time + T). A short control cycle is necessary to allow the system to react quickly to new job submissions and changes in the task length observed for running jobs. In each cycle, the Placement Algorithm component examines the placement of tasks on TaskTrackers and their resource allocations, evaluates different candidate placement matrices and proposes the final output placement to be enforced until next control cycle. The Job Utility Calculator calculates a utility value for an input placement matrix which is then used by the Placement Algorithm to choose the best placement choice available.

The **Task Scheduler** is responsible for enforcing the placement decisions, and for moving the system smoothly between a placement decision made in the last cycle to a new decision produced in the most recent cycle. The Task Scheduler schedules tasks according to

the placement decision made by the Placement Controller. Whenever a task completes, it is the responsibility of the Task Scheduler to select a new task to execute in the freed slot, by providing a task of the appropriate type from the appropriate job to the given TaskTracker.

The following sections will concentrate on the problem solved by the Placement Algorithm component in a single control cycle.

4.2.3 *Performance Model*

To measure the performance of a job given a placement matrix, we define a utility function that combines the number of map and reduce slots allocated to the job with its completion time goal and job characteristics. Below we provide a description of this function.

Given placement matrices P^M and P^R, we can define the number of map and reduce slots allocated to a job j as $s^j_{alloc} = \sum_{tt \in \mathcal{TT}} P^M_{j,tt}$ and $r^j_{alloc} = \sum_{tt \in \mathcal{TT}} P^R_{j,tt}$ correspondingly.

Based on these parameters and the previous definitions of s^j_{pend} and r^j_{pend}, we define the utility of a job j given a placement P as:

$$u_j(P) = u^M_j(P^M) + u^R_j(P^R), \quad \text{where } P = P^M + P^R \tag{5}$$

where u^M_j is a utility function that denotes increasing satisfaction of a job given a placement of map tasks, and u^R_j is a utility function that shows satisfaction of a job given a placement of reduce tasks. The definition of both functions is:

$$u^M_j(P^M) = \begin{cases} \frac{s^j_{alloc} - s^j_{req}}{s^j_{pend} - s^j_{req}} & s^j_{alloc} \geq s^j_{req} \\ \frac{log(s^j_{alloc})}{log(s^j_{req})} - 1 & s^j_{alloc} < s^j_{req} \end{cases} \tag{6}$$

$$u^R_j(P^R) = \frac{log(r^j_{alloc})}{log(r^j_{pend})} - 1 \tag{7}$$

Notice that in practice a job will never get more tasks allocated to it than it has remaining: to reflect this in theory we cap the utility at $u_j(P) = 1$ for those cases.

The definition of u differentiates between two cases: (1) the satisfaction of the job grows logarithmically from $-\infty$ to 0 if the job has fewer map slots allocated to it than it requires to meet its completion time goal; and (2) the function grows linearly between 0 and 1, when $s^j_{alloc} = s^j_{pend}$ and thus all pending map tasks for this job are allocated a slot in the current control cycle. Notice that u^M_j is a monotonically

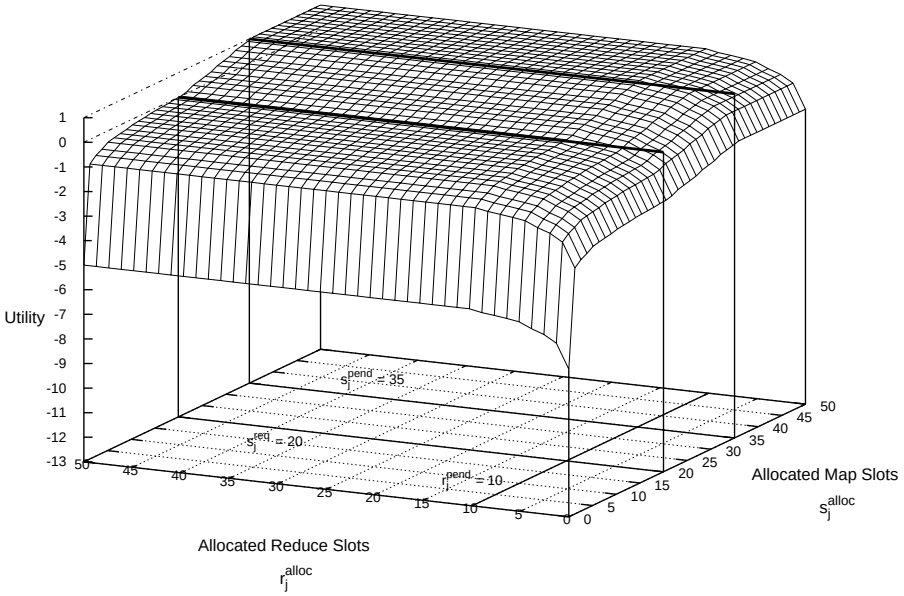


Figure 4.2: Shape of the Utility Function when $s^j_{req} = 20$, $s^j_{pend} = 35$, and $r^j_{pend} = 10$

increasing utility function, with values in the range $(-\infty, 1]$. The intuition behind this function is that a job is unsatisfied ($u^M_j < 0$) when the number of slots allocated to map tasks is less than the minimum number required to meet the completion time goal of the job. Furthermore, the logarithmic shape of the function stresses the fact that it is critical for a job to make progress and therefore at least one slot must be allocated. A job is no longer unsatisfied ($u^M_j = 0$) when the allocation equals the requirement ($s^j_{alloc} = s^j_{req}$), and its satisfaction is positive ($u^M_j > 0$) and grows linearly when it gets more slots allocated than required. The maximum satisfaction occurs when all the pending tasks are allocated within the current control cycle ($s^j_{alloc} = s^j_{pend}$). The intuition behind u^R_j is that reduce tasks should start at the earliest possible time, so the shuffle sub-phase of the job (reducers pulling data produced by map tasks) can be fully pipelined with execution of map tasks. The logarithmic shape of this function indicates that any placement that does not run all reducers for a running job is unsatisfactory. The range of this function is $[-1, 0]$ and, therefore, it is used to subtract satisfaction of a job that, independently of the placement of map tasks, has unsatisfied demand for reduce tasks. If all the reduce tasks for a job are allocated, this function gets value 0 and thus, $u_j(P) = u^M_j(P^M)$.

Figure 4.2 shows the generic shape of the utility function for a job that requires at least 20 map tasks to be allocated concurrently ($s^j_{req} = 20$) to meet its completion time goal, has 35 map tasks ($s^j_{pend} = 35$) pending to be executed, and has been configured to run 10 reduce

tasks ($r^j_{pend} = 10$), none of which have been started yet. On the X axis, a variable number of allocated slots for reduce tasks (r^j_{alloc}) is shown. On the Y axis, a variable number of allocated slots for map tasks (s^j_{alloc}) is shown. Finally, the Z axis shows the resulting utility value.

4.2.4 *Placement Algorithm and Optimization Objective*

Given an application placement matrix P, a utility value can be calculated for each job in the system. The performance of the system can then be measured as an ordered vector of job utility values, U. The objective of RAS is to find a new placement P of jobs on TaskTrackers that maximizes the global objective of the system, $U(P)$, which is expressed as follows:

$$\max \quad \min_j u_j(P) \tag{8}$$

$$\min \quad \Omega_{tt,r} - \sum_{tt}(\sum_j P_{j,tt}) * \Gamma_{j,r} \tag{9}$$

such that

$$\forall_{tt}\forall_r \quad (\sum_j P_{j,tt}) * \Gamma_{j,r} \leq \Omega_{tt,r} \tag{10}$$

This optimization problem is a variant of the Class Constrained Multiple-Knapsack Problem. Since this problem is NP-hard, the scheduler adopts a heuristic inspired by [75], and which is outlined in Algorithm 1. The proposed algorithm consists of two major steps: placing reduce tasks and placing map tasks.

Reduce tasks are placed first to allow them to be evenly distributed across TaskTrackers. By doing this we allow reduce tasks to better multiplex network resources when pulling intermediate data and also enable better storage usage. The placement algorithm distributes reduce tasks evenly across TaskTrackers while avoiding collocating any two reduce tasks. If this is not feasible—due to the total number of tasks—it then gives preference to avoiding collocating reduce tasks from the same job. Recall that in contrast to other existing schedulers, RAS dynamically adjusts the number of map and reduce tasks allocated per TaskTracker while respecting its resource constraints. Notice also that when reduce tasks are placed first, they start running in shuffle phase, so that their demand of resources is directly proportional to the number of map tasks placed for the same job. Therefore, in the absence of map tasks for the same job, a reduce task in shuffle phase only consumes memory. It therefore follows that the system is unlikely to be fully booked by reduce tasks.

Algorithm 1 Placement Algorithm run at each Control Cycle

Inputs $P^M(job,tt)$: Placement Matrix of Map tasks, $P^R(job,tt)$: Placement Matrix of Reduce tasks, J: List of Jobs in the System, D: Resource demand profile for each job, TT: List of TaskTrackers in the System
Γ_j and Ω_{tt}: Resource demand and capacity for each Job each TaskTracker correspondingly, as used by the auxiliary function $room\_for\_new\_job\_slot$

{——————— **Place Reducers** ———————}

1: **for** *job* in J **do**
2: Sort TT in increasing order of overall number of reduce tasks placed (first criteria), and increasing order of number of reducers *job* placed (second criteria)
3: **for** tt in TT **do**
4: **if** $room\_for\_new\_job\_slot(job,tt)$ & $r^{job}_{pend} > 0$ **then**
5: $P^R(job,tt) = P^R(job,tt) + 1$
6: **end if**
7: **end for**
8: **end for**

{——————— **Place Mappers** ———————}

9: **for** $round = 1 \ldots rounds$ **do**
10: **for** tt in TT **do**
11: $job_{in} \leftarrow min\ U(job_{in},P)$, $room\_for\_new\_job\_slot(job_{in},tt)$,
12: $job_{out} \leftarrow max\ U(job_{out},P)$, $P^M(job_{out},tt) > 0$
13: **repeat**
14: $P_{old} \leftarrow P$
15: $job_{out} \leftarrow max\ U(job_{out},P)$, $P(job_{out},tt) > 0$
16: $P^M(job_{out},tt) = P^M(job_{out},tt) - 1$
17: $job_{in} \leftarrow min\ U(job_{in},P)$, $room\_for\_new\_job\_slot(job_{in},tt)$
18: **until** $U(job_{out},P) < U(job_{in},P_{old})$
19: $P \leftarrow P_{old}$
20: **repeat**
21: $job_{in} \leftarrow min\ U(job_{in},P)$, $room\_for\_new\_job\_slot(job_{in},tt)$
22: $P^M(job_{in},tt) = P^M(job_{in},tt) + 1$
23: **until** $\nexists job$ such that $room\_for\_new\_job\_slot(job,tt)$
24: **end for**
25: **end for**
26: **if** map phase of a job is about to complete in this control cycle **then**
27: switch profile of placed reducers from shuffle to reduce and wait for Task Scheduler to drive the transition.
28: **end if**

The second step is placing map tasks. This stage of the algorithm is utility-driven and seeks to produce a placement matrix that balances satisfaction across jobs while treating all jobs fairly. This is achieved by maximizing the lowest utility value in the system. This part of the algorithm executes a series of rounds, each of which tries to improve the lowest utility of the system. In each round, the algorithm removes allocated tasks from jobs with the highest utility, and allocates more tasks to the jobs with the lowest utility. For the sake of fairness, a task gets de-allocated only if the utility of its corresponding job remains higher than the lowest utility of any other job in the system. This results in increasing the lowest utility value across jobs in every round. The loop stops after a maximum number of rounds has reached, or until the system utility no longer improves. This process allows for satisfying the optimization objective introduced in Equation 8.

Recall that RAS is resource-aware and hence all decisions to remove and place tasks are made considering the resource constraints and demands in the system. Furthermore, in order to improve system utilization it greedily places as many tasks as resources allow. This management technique is novel and allows for satisfying the optimization objective introduced in Equation 9.

The final step of the algorithm is to identify if any running jobs will complete their map phase during the current control cycle. This transition is important because it implies that reduce tasks for those jobs will start the reduce phase. Therefore, the algorithm has to switch the resource demand profile for the reduce tasks from 'shuffle' to 'reduce'. Notice that this change could overload some TaskTrackers in the event that the 'reduce' phase of the reduce tasks uses more resources than the 'shuffle' phase. RAS handles this by having the Task Scheduler drive the placement transition between control cycles, and provides overload protection to the TaskTrackers.

4.2.5 *Task Scheduler*

The Task Scheduler drives transitions between placements while ensuring that the actual demand of resources for the set of tasks running in a TaskTracker does not exceed its capacity. The placement algorithm generates new placements, but these are not immediately enforced as they may overload the system due to tasks still running from the previous control cycle. The Task Scheduler component takes care of transitioning without overloading any TaskTrackers in the system by picking jobs to assign to the TaskTracker that do not exceed its current capacity, sorted by lowest utility first. For instance, a TaskTracker that is running 2 map tasks of job A may have a different assignment for the next cycle, say, 4 map tasks of job B. Instead of starting the new tasks right away while the previous ones are still running, new tasks will only start running as previous tasks complete

and enough resources are freed. Recall that the scheduler is adaptive as it continuously monitors the progress of jobs and their average task length, so that any divergence between the placement matrix produced by the algorithm and the actual placement of tasks enforced by the Task Scheduler component is noticed and considered in the following control cycle. The Task Scheduler component is responsible for enforcing the optimization objective shown in Equation 10.

4.2.6 *Job Profiles*

The proposed job scheduling technique relies on the use of job profiles containing information about the resource consumption for each job. Profiling is one technique that has been successfully used in the past for MapReduce clusters. Its suitability in these clusters stems from the fact that in most production environments jobs are ran periodically on data corresponding to different time windows [79]. Hence, profiles remains fairly stable across runs [82].

Our profiling technique works offline. To build a job profile we run a job in a sandbox environment with the same characteristics of the production environment. We run the job in isolation multiple times in the sandbox using different configurations for the number of map task slots per node (1 map, 2 maps, ..., up to N). The number of reduce tasks is set to the desired level by the user submitting the job. In the case of multiple reduce tasks, they execute on different nodes.

From the multiple configurations, we select that one in which the job completed fastest, and use that execution to build our profile. We monitor CPU, I/O and memory usage in each node for this configuration using `vmstat`. The reasoning behind this choice is that we want to monitor the execution of a configuration in which competition for resources occurs and some system bottlenecks are hit, but in which severe performance degradation is not yet observed.

Note that obtaining CPU and memory is straight forward for the various phases. For example, if the bottleneck is CPU (that is to say, the node experiences 100% CPU utilization) and there are 4 map tasks running, each map task consumes 25% CPU. Profiling I/O in the shuffle phase is less trivial. Each reduce task has a set of threads responsible for pulling map outputs (intermediate data generated by the map tasks): the number of these threads is a configurable parameter in Hadoop (hereafter `parallelCopies`). These threads are informed about the availability and location of a new map output whenever a map task completes. Consequently, independent of the number of map outputs available, the reduce tasks will never fetch more than `parallelCopies` map outputs concurrently. During profiling we ensure that there are at least `parallelCopies` map outputs available for retrieval and we measure the I/O utilization in the reduce task while

shuffling. It can therefore be seen that our disk I/O measurement is effectively an upper bound on the I/O utilization of the shuffle phase.

In RAS we consider jobs that run periodically on data with uniform characteristics but different sizes. Since the map phase processes a single input split of fixed size and the shuffle phase retrieves `parallelCopies` map outputs concurrently (independently of the input data size) their resource profile remain similar. Following these observations, the completion time of the map tasks remains the same while the completion time of the shuffle phase may vary depending on the progress rate of the map phase. The case of the reduce phase is more complicated. The reducer phase processes all the intermediate data at once and this one tends to increase (for most jobs we know of) as the input data size increases. In most of the jobs that we consider we observe that the completion time of the reduce phase scales linearly. However, this is not always the case. Indeed, if the job has no reduce function and simply relies on the shuffle phase to sort, we observe that the completion time scales super-linearly ($n \times log(n)$). Having said that, our approach can be improved, for example by using historical information.

Profile accuracy plays a role in the performance of RAS. Inaccurate profiles lead to resource under- or overcommitment. This dependency exists in a slot-based system too, as it also requires some form of profiling to determine the optimal number of slots. The optimal slot number measures a job-specific capacity of a physical node determined by a bottleneck resource for the job, and it can be easily converted into an approximate resource profile for the job (by dividing bottleneck resource capacity by the slot number). Provided with these profiles, RAS allows jobs with different optimal slot numbers to be co-scheduled, which is a clear improvement over classical slot-based systems. The profiling technique used in this chapter allows multi-resource profiles to be built, which helps improve utilization when scheduling jobs with different resource bottlenecks. Since the sandbox-based method of profiling assumes that resource utilization remains stable among different runs of the same job on different data, it may fail to identify a correct profile for jobs that do not meet this criterion. For those jobs, an online profiler or a hybrid solutions with reinforcement learning may be more appropriate since RAS is able to work with profiles that change dynamically and allows different profiling technologies to be used for different jobs. While not addressing in this chapter, such techniques have been studied in [38, 54, 76].

4.3 EVALUATION

In this section we include results from two experiments that explore the two objectives of RAS: improving resource utilization in the cluster (Experiment 1) and meeting jobs' completion time goals (Exper-

iment 2). In Experiment 1, we consider resource utilization only, and compare RAS with a state-of-the-art non-resource-aware Hadoop scheduler. In order to gain insight on how RAS improves resource utilization, we set a relaxed completion time goal with each job. This allow us to isolate both objectives and reduce the effect of completion time goals in the algorithm. In Experiment 2, we consider completion time goals on the same workload. Thus effectively evaluating all capabilities of RAS.

4.3.1 *Experimental Environment and Workload*

We perform all our experiments on a Hadoop cluster consisting of 22 2-way 64-bit 2.8GHz Intel Xeon machines. Each machine has 2GB of RAM and runs a 2.6.17 Linux kernel. All machines in the cluster are connected via a Gigabit Ethernet network. The version of Hadoop used is 0.23. The cluster is configured such that one machine runs the JobTracker, another machine hosts the NameNode, and the remaining 20 machines each host a DataNode and a TaskTracker.

To evaluate RAS we consider a representative set of applications included in the Gridmix benchmark, which is part of the Hadoop distribution. These applications are Sort, Combine and Select. For each application we submit 3 different instances with different input sizes, for a total of 9 jobs in each experiment. A summary of the workload can be seen in Table 4.1, including the label used for each instance in the experiments, the size of its associated input data set, the submission time, and the time taken by each job to complete if the entire experimental cluster is dedicated to it. Additionally, we include the actual completion times observed for each instance in Experiment 1 and 2. Finally, for Experiment 2, we include also the completion time goal associated to each instance.

The resource consumption profiles provided to the scheduler are shown in Table 4.2. They were obtained following the description provided in Section 4.2.6. The values are the percentage of each TaskTracker's capacity that is used by a single execution of the sub-phase in question.

4.3.2 *Experiment 1: Execution with relaxed completion time goals*

The goal of this experiment is to evaluate how RAS improves resource utilization compared to the Fair Scheduler when completion time goals are so relaxed that the main optimization objective of the algorithm is to maximize resource utilization (see Equation 17). To this end we associate with each job instance an highly relaxed completion time goal. We run the same workload using both the Fair Scheduler and RAS and compare different aspects of the results.

	SORT			COMBINE			SELECT		
Instance label	J1	J8	J9	J2	J6	J7	J3	J4	J5
Input size (GB)	90	19	6	5	13	50	10	25	5
Submission time (s)	0	2,500	3,750	100	600	1,100	200	350	500
Length in isolation (s)	2,500	350	250	500	750	2,500	400	280	50
EXPERIMENT 1									
Completion time (s)	3,113	3,670	4,100	648	3,406	4,536	1,252	608	623
EXPERIMENT 2									
Completion time (s)	3,018	3,365	4,141	896	2,589	4,614	802	550	560
Completion time goal (s)	3,000	3,400	4,250	850	2,600	6,000	1,250	1,100	950

Table 4.1: Workload characteristics: 3 Applications, 3 Job instances each (Big, Medium, and Small)

	SORT	COMBINE	SELECT
	Map/Shuffle/Reduce	Map/Shuffle/Reduce	Map/Shuffle/Reduce
CPU	30%/-/20%	25%/-/10%	15%/-/10%
I/O	45%/0.15%/50%	10%/0.015%/10%	20%/0.015%/10%
Memory	25%/-/60%	10%/-/25%	10%/-/25%

Table 4.2: Job profiles (shuffle: consumed I/O per map placed, upper bound set by `parallelCopies`, the number of threads that pull map output data)

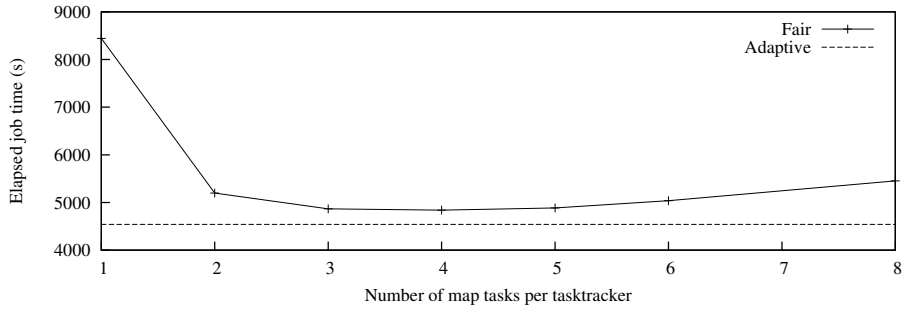
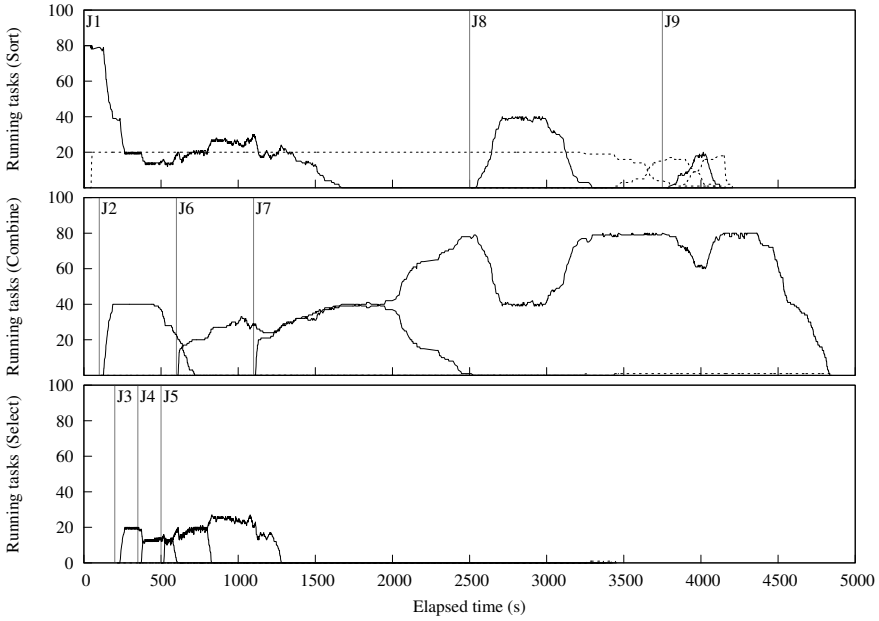


Figure 4.3: Experiment 1: Workload makespan with different Fair Scheduler configurations (Y-axis starts at 4000 seconds)

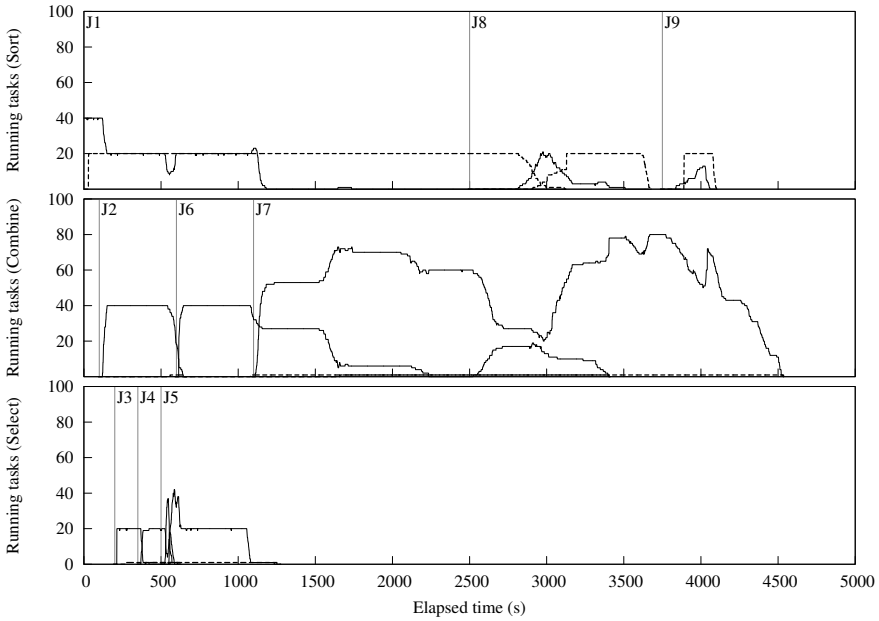
Dynamic task concurrency level per TaskTracker. Our first objective in this experiment is to study how the dynamic management of the level of task concurrency per-TaskTracker improves workload performance. To this end, we run the same workload using the Fair Scheduler with different concurrency level configurations: specifically, we vary the maximum number of map slots per TaskTracker from 1 to 8, and compare the results with the execution using RAS. Results are shown in Figure 4.3. As can be seen, the best static configuration uses 4 concurrent map tasks per TaskTracker (80 concurrent tasks across 20 TaskTrackers). Configurations that result in low and high concurrency produce worse makespan due to resources being underutilized and overcommitted, respectively.

We observe that RAS outperforms the Fair Scheduler for all configurations, showing an improvement that varies between 5% and 100%. Our traces show that the average task concurrency level in RAS was 3.4 tasks per TaskTracker. Recall that the best static configuration of per-TaskTracker task concurrency depends on the workload characteristics. As workloads change over time in real systems, even higher differences between static and dynamic management would be observed. RAS overcomes this problem by dynamically adapting task concurrency level based on to the resource usage in the system.

Resource allocation and Resource utilization. Now we look in more detail at the execution of the workload using RAS as compared to the Fair Scheduler running a static concurrency level of 4. Figures 4.4a and 4.4b show the task assignment resulting from both schedulers. For the sake of clarity, we group jobs corresponding to the same application (Sort, Combine or Select) into different rows. Each row contains solid and dotted lines, representing the number of running map and reduce tasks respectively. The submission time for each job is shown by a (labeled) vertical line, following the convention presented in Table 4.1. Combine and Select are configured to run one single reduce task per job since there is no benefit from running them with more reduce tasks on our testing environment; the dotted line representing the reduce is at the bottom of the chart. As it can be observed, RAS does not allocate more concurrent map slots than the Fair Scheduler during most of the execution. Moreover, the sum of reduce and map tasks remains lower than the sum of reduce and map tasks allocated by the Fair Scheduler except for a small time interval ($\sim$ 100s) immediately after the submission of Job 6 (J6). RAS is able to improve the makespan while maintaining a lower level of concurrency because it does better at utilizing resources which ultimately results in better job performance. To get a better insight on how RAS utilizes resources as compared to the Fair Scheduler we plot the CPU utilization for both schedulers in Figures 4.5a and 4.5b. These figures show the percentage of CPU time that TaskTrackers spent running tasks (either in system or user space), and the time that the CPU was

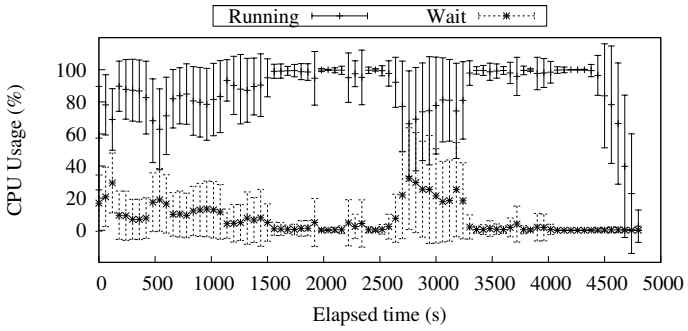


(a) Fair Scheduler

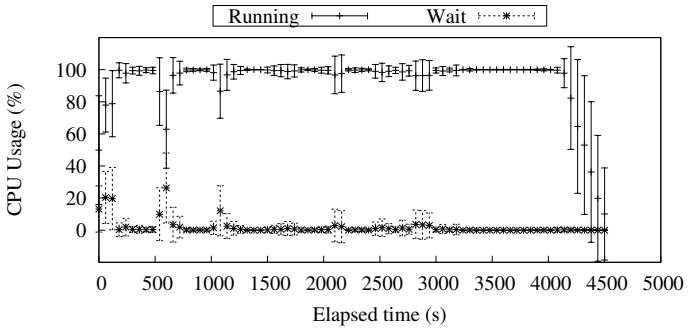


(b) RAS

Figure 4.4: Experiment 1: Workload execution: (a) corresponds to Fair Scheduler using 4 slots per TaskTracker, and (b) corresponds to RAS using a variable number of slots per TaskTracker



(a) Fair Scheduler



(b) RAS

Figure 4.5: Experiment 1: CPU utilization: (a) corresponds to Fair Scheduler using 4 slots per TaskTracker, and (b) corresponds to RAS using a variable number of slots per TaskTracker

waiting. For each metric we show the mean value for the cluster, and the standard deviation across TaskTrackers. Wait time represents the time that the CPU remains idle because all threads in the system are either idle or waiting for I/O operations to complete. Therefore, it is a measure of resource wastage, as the CPU remains inactive. While wait time is impossible to avoid entirely, it can be reduced by improving the overlapping of tasks that stress different resources in the TaskTracker. It is noticeable that in the case of the Fair Scheduler the CPU spends more time waiting for I/O operations to complete than RAS. Further, modifying the number of concurrent slots used by the Fair Scheduler does not improve this result. The reason behind this observation is key to our work: other schedulers do not consider the resource consumption of applications when making task assignment decisions, and therefore are not able to achieve good overlap between I/O and CPU activity.

Utility guidance. Finally, to illustrate the role of the utility function in RAS, Figure 4.6 shows the utility value associated with each job during the execution of the workload. Since the jobs have extremely lax completion time goals, they are assigned a utility value above 0 immediately after one task for each job is placed. As can be seen, the allocation algorithm balances utility values across jobs for most of the execution time. In some cases, though, a job may get higher util-

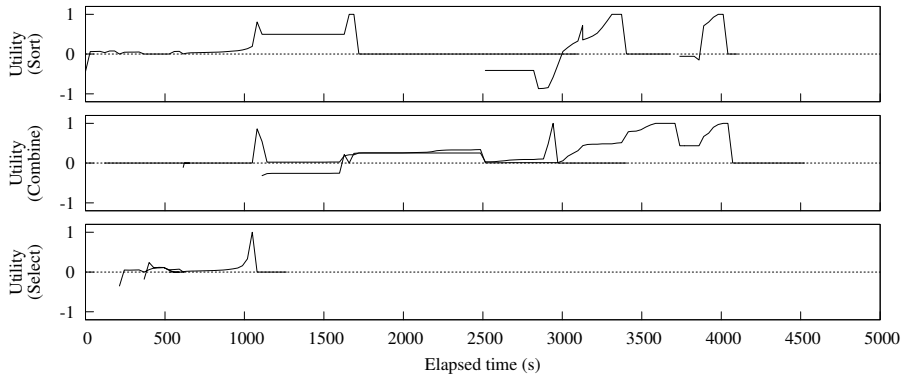


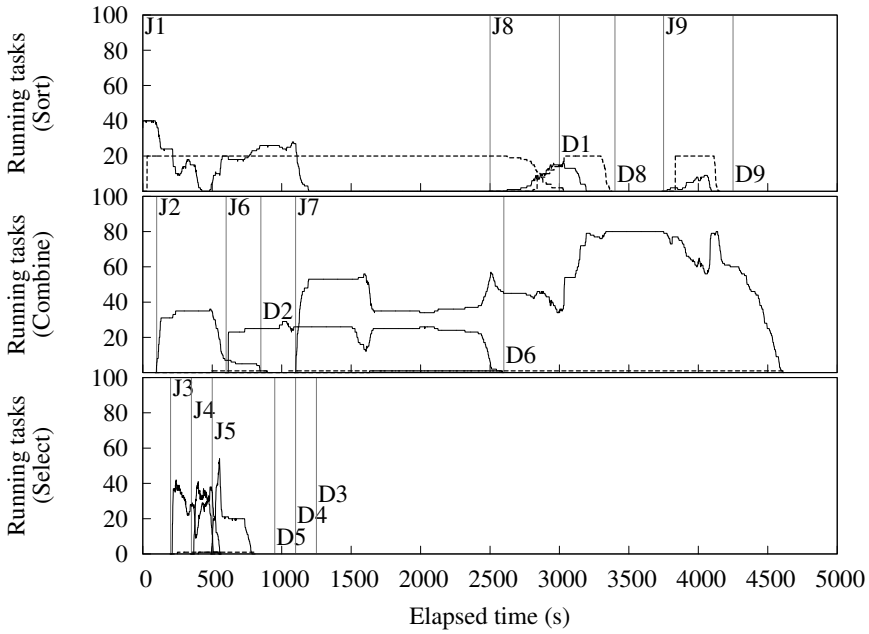
Figure 4.6: Experiment 1: Job Utility

ity than the others: this is explained by the fact that as jobs get closer to completion, the same resource allocation results in higher utility. This is seen in our experiments: for all jobs, the utility increases until all their remaining tasks are placed. In this experiment we can also see that Job 7 has a very low utility right after it is launched (1,100s) in contrast with the relatively high utility of Job 1, even though most resources are actually assigned to Job 7. This is because while Job 1 has very few remaining tasks, no tasks from Job 7 have been completed and thus its resource demand estimation is not yet accurate. This state persists until approximately time 1,650s).

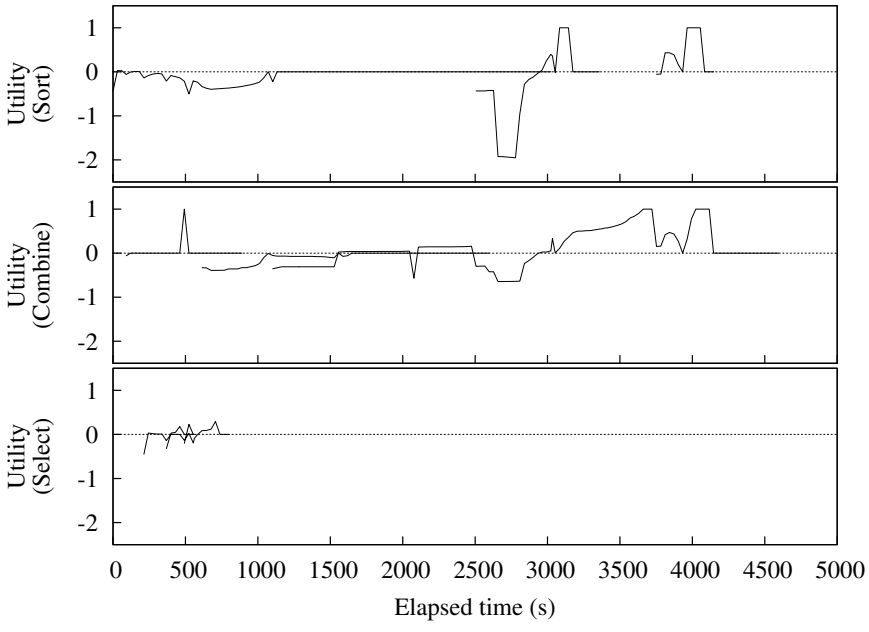
4.3.3 *Experiment 2: Execution with tight completion time goals*

In this experiment we evaluate the behavior of RAS when the applications have stringent completion time goals. To do this we associate a tight completion time goal with the workload described for our previous experiment.

Figure 4.7a shows the number of concurrent tasks allocated to each job during the experiment. We use vertical lines and labels to indicate submission times (labeled J1 to J9) and completion time goals (labeled D1 to D9) for each of the nine jobs in the workload. To illustrate how RAS manages the tradeoff between meeting completion time goals and maximizing resource utilization, we look at the particular case of Job 1 (Sort), Job 7 (Combine) and Job 8 (Sort), submitted at times J1 (0s), J7 (1,100s) and J8 (2,500s) respectively. In Experiment 1 their actual completion times were 3,113s, 4,536s and 3,670s, while in Experiment 2 they completed at times 3,018s, 4,614s and 3,365s respectively. Because their completion time goals in Experiment 2 are 3,000s, 6,000s and 3,400s (a factor of 1.2X, 1.9X and 2.5X compared to their length observed in isolation), the algorithm allocates more tasks to Job 1 and Job 8 at the expense of Job 7, which sees its actual completion time delayed with respect to Experiment 1 but still makes its more relaxed goal. It is important to remark again that completion



(a)



(b)

Figure 4.7: Experiment 2: Workload execution and Job utility

time goals in our scheduler are soft deadlines used to guide the workload management as opposed to strict deadlines in which missing a deadline is associated with strong penalties. Finally, notice that Job 1 and Job 8 would have clearly missed their goals in Experiment 1: here, however, RAS adaptively moves away from the optimal placement in terms of resource allocation to adjust the actual completion times of jobs. Recall that RAS is still able to leverage a resource model while aiming at meeting deadlines, and still outperforms the best configuration of Fair Scheduler by 167 seconds, 4,781s compared to 4,614s.

To illustrate how utility is driving placement decisions, we include Figure 4.7b, which shows the utility of the jobs during the workload execution and gives a better intuition of how the utility function drives the scheduling decisions. When a job is not expected to reach its completion time goal with the current placement, its utility value goes negative. For instance, starting from time 2,500s when J8 is launched and the job still has very few running tasks, the algorithm places new tasks to J8 at the expense of J7. However, as soon as J8 is running the right amount of tasks to reach the deadline, around time 3,000s, both jobs are balanced again and the algorithm assigns more tasks to J7.

4.4 RELATED WORK

Much work have been done in the space of scheduling for MapReduce. Since the number of slots in a Hadoop cluster is fixed through out the lifetime of the cluster, most of the proposed solutions can be reduced to a variant of the *task-assignment* or *slot-assignment* problem. The Capacity Scheduler [88] is a pluggable scheduler developed by Yahoo! which partition resources into pools and provides priorities for each pool. Hadoop's Fair Scheduler [89] allocates equal shares to each tenant in the cluster. Quincy scheduler [42] proposed for the Dryad environment [40] also shares similar fairness goals. All these schedulers are built on top of the slot model and do not support user-level goals.

The performance of MapReduce jobs has attracted much interest in the Hadoop community. Stragglers, tasks that take an unusually long time to complete, have been shown to be the most common reason why the total time to execute a job increases [24]. Speculative scheduling has been widely adopted to counteract the impact of stragglers [24, 89]. Under this scheduling strategy, when the scheduler detects that a task is taking longer than expected it spawns multiple instances of the task and takes the results of the first completed instance, killing the others [89]. In Mantri [5] the effect of stragglers is mitigated via the 'kill and restart' of tasks which have been identified as potential stragglers. The main disadvantage of these techniques is that killing and duplicating tasks results in wasted resources [89, 5]. In

RAS we take a more proactive approach, in that we prevent stragglers resulting from resource contention. Furthermore, stragglers caused by skewed data cannot be avoided at run-time [5] by any existing technique. In RAS the slow-down effect that these stragglers have on the end-to-end completion time of their corresponding jobs is mitigated by allocating more resources to the job so that it can still complete in a timely manner.

Recently, there has been increasing interest in user-centric data analytics. One of the seminal works in this space is [59]. In this work, the authors propose a scheduling scheme that enables soft-deadline support for MapReduce jobs. It differs from RAS in that it does not take into consideration the resources in the system. Flex [86] is a scheduler proposed as an add-on to the Fair Scheduler to provide Service-Level-Agreement (SLA) guarantees. More recently, ARIA [82] introduces a novel resource management framework that consists of a job profiler, a model for MapReduce jobs and a SLO-scheduler based on the Earliest Deadline First scheduling strategy. Flex and Aria are both slot-based and therefore suffers from the same limitations we mentioned earlier. One of the first works in considering resource awareness in MapReduce clusters is [26]. In this work the scheduler classifies tasks into good and bad tasks depending on the load they impose in the worker machines. More recently, the Hadoop community has also recognized the importance of developing a resource-aware scheduling for MapReduce. [12] outlines the vision behind the Hadoop scheduler of the future. The framework proposed introduces a resource model consisting of a 'resource container' which is—like our 'job slot'—fungible across job tasks. We think that our proposed resource management techniques can be leveraged within this framework to enable better resource management.

4.5 SUMMARY

This chapter presents a resource-aware scheduling technique for MapReduce multi-job workloads that aims at improving resource utilization across machines while observing completion time goals. Existing MapReduce schedulers define a static number of slots to represent the capacity of a cluster, creating a fixed number of execution slots per machine. This abstraction works for homogeneous workloads, but fails to capture the different resource requirements of individual jobs in multi-user environments. The proposed technique leverages job profiling information to dynamically adjust the number of slots on each machine, as well as workload placement across them, to maximize the resource utilization of the cluster. In addition, this technique is also guided by user-provided completion time goals for each job.

The work described in this chapter is based on the following main publication:

[63] Jordà Polo, Claris Castillo, David Carrera, Yolanda Becerra, Ian Whalley, Malgorzata Steinder, Jordi Torres, and Eduard Ayguadé. Resource-Aware Adaptive Scheduling for MapReduce Clusters. In *ACM IFIP USENIX 12th International Middleware Conference*, pages 187–207, Lisbon, Portugal, 2011. Springer. ISBN 978-3-642-25820-6. doi: 10.1007/978-3-642-25821-3_10

5

SCHEDULING WITH SPACE AND TIME CONSTRAINTS IN SHARED ENVIRONMENTS

5.1 INTRODUCTION

This chapter focuses on a scenario that is becoming increasingly important in data centers. Instead of running on dedicated machines, MapReduce is executed along with other resource-consuming workloads, such as transactional applications. All workloads may potentially share the same data store, some of them consuming data for analytics while others may be acting as data generators. Twitter, Facebook, and other companies that need to handle large amounts of data, accessing and processing it in different ways and for different purposes, follow this approach of sharing multiple workloads on the same data center.

These shared environments involve higher workload consolidation, which helps improve resource utilization, but is also challenging due to the interaction between workloads of very different nature. One of the major issues found in this scenario is related to the integration of the storage. Storage is a key component since it usually deals with multiple producers and consumers of data, and often serves different kinds of workloads at the same time: from responding to transactional queries to storing the output of long-running data analytics jobs, each one of them with slightly different needs.

There are also other issues that arise when multiple workloads are collocated sharing the same machines. MapReduce schedulers, for instance, assume that the amount of available resources remains the same over time, but resources are no longer stable in a shared environment with transactional workloads, which are known to be bursty and have a varying demand over time. Hence, this scenario requires

deep coordination between management components, and single applications can not be considered in isolation but in the full context of mixed workloads in which they are deployed.

This chapter is focused on two related problems found in shared environments with MapReduce. First, Section 5.2 addresses one of the issues found in shared environments where multiple workloads may use the same data store for different purposes. In particular, the proposal uses a distributed key-value store as a good compromise between traditional databases and distributed filesystems, and enables it with the necessary snapshotting mechanisms to be used by both transactional and analytics workloads. This contribution is presented first since it establishes and validates the scenario. Later, Section 5.3 introduces a scheduler and a performance model for MapReduce in shared environments. The proposed scheduler aims to improve resource utilization across machines while observing completion time goals, taking into account the resource demands of non-MapReduce workloads, and assuming that the amount of resources made available to the MapReduce applications is dynamic and variable over time. This is achieved thanks to a new algorithm that provides a more proactive approach for the scheduler to estimate the need of resources that should be allocated to each job.

5.2 ENABLING DISTRIBUTED KEY-VALUE STORES WITH SNAPSHOT SUPPORT

Current distributed key-value stores generally provide greater scalability at the expense of weaker consistency and isolation. However, additional isolation support is becoming increasingly important in the environments in which these stores are deployed, where different kinds of applications with different needs are executed, from transactional workloads to data analytics. While fully-fledged ACID support may not be feasible, it is still possible to take advantage of the design of these data stores, which often include the notion of multiversion concurrency control, to enable them with additional features at a much lower performance cost and maintaining its scalability and availability. This section explores the effects that additional consistency guarantees and isolation capabilities may have on a state of the art key-value store: Apache Cassandra. We propose and implement a new multiversioned isolation level that provides stronger guarantees without compromising Cassandra's scalability and availability. As shown in our experiments, our version of Cassandra allows Snapshot Isolation-like transactions, preserving the overall performance and scalability of the system.

5.2.1 *Introduction*

In recent years, the industry and research community have witnessed an extraordinary growth in research and development of data-analytic technologies. In addition to distributed, large-scale data processing with models like MapReduce, new distributed data stores have been introduced to deal with huge amounts of structured and semi-structured data: Google's BigTable [19], Amazon's Dynamo [25], and others often modeled after them. These key-value data stores were created out of need for highly reliable and scalable databases, and they have been extremely successful in introducing new ways to think about large-scale models and help solve problems that require dealing with huge amounts of data.

The emergence of these new data stores, along with its widespread and rapid adoption, is changing the way we think about storage. Only a few years ago, relational database systems used to be the only back-end storage solution, but its predominant and almost exclusive position is now being challenged. While scalable key-value stores are definitely not a replacement for RDBMs, which still provide a richer set of features and stronger semantics, they are marking an important shift in storage solutions. Instead of using a single database system on a high-end machine, many companies are now adopting a number of different and complementary technologies, from large-scale data processing frameworks to key-value stores to relational databases, often running on commodity hardware or cloud environments.

This new scenario is challenging since key-value stores are being adopted for uses that were not initially considered, and data must sometimes be accessed and processed with a variety of tools as part of its dataflow. In this environment, distributed key-value stores are becoming one of the corner-stones as they become the central component of the back-end, interacting concurrently with multiple producers and consumers of data, and often serving different kinds of workloads at the same time: from responding to transactional queries to storing the output of long-running data analytics jobs.

Consistency and isolation become increasingly important as soon as multiple applications and workloads with different needs interact with each other. Providing strong semantics and fully-fledged transactions on top of distributed key-value stores often involves a significant penalty on the performance of the system since it is orthogonal to its goals. So, while fully-fledged ACID support may not be feasible, it is still possible to take advantage of the design of these data stores, which often include the notion of multiversion concurrency control, to enable them with additional features at a much lower performance cost and maintaining its scalability and availability.

This is the approach we are following here. Our goal is to provide stronger isolation on top of a distributed key-value store in order

to allow certain operations that would otherwise not be possible or require significant effort on the client side, but without compromising its performance. We implement this improved isolation level in the form of readable snapshots on top of Apache Cassandra, a state of the art distributed column-oriented key-value store.

The following sections describe our approach and implementation. Section 5.2.2 describes isolation and consistency levels in Cassandra. Section 5.2.3 describes how we have extended the level of isolation and how we implement it on top of Cassandra. An evaluation of our implementation is studied in Section 5.2.4. Finally, Section 5.2.5 discusses the related work.

5.2.2 *Isolation and Consistency Levels*

The ultimate goal of current distributed key-value stores such as Cassandra [45] is similar to other database systems, reading and writing data operations, but with a stronger focus on adapting to the increased demands of large-scale workloads. While traditional databases provide strong consistency guarantees of replicated data by controlling the concurrent execution of transactions, Cassandra provides tunable consistency in order to favour scalability and availability. While there is no tight control of the execution of concurrent transactions, Cassandra still provides mechanisms to resolve conflicts and provide durability even in the presence of node failures.

Traditionally, database systems have provided different isolation levels that define how operations are visible to other concurrent operations. Standard ANSI SQL isolation levels have been criticized as too few [16], but in addition to standard ANSI SQL, other non-standard levels have been widely adopted by database systems. One such level is Snapshot Isolation, which guarantees that all reads made within a transaction see a consistent version of the data (a *snapshot*).

Cassandra, on the other hand, unlike traditional databases, does not provide any kind of server-side transaction or isolation support. For instance, if an application needs to insert related data to multiple tables, additional logic will be needed on the application (e.g. to manually roll-back the changes if one operation fails). Instead, Cassandra provides a tunable consistency mechanism that defines the state and behaviour of the system after executing an operation, and basically allows specifying how much consistency is required for each query.

Tables 5.1 and 5.2 show Cassandra's tunable read and write consistency levels, respectively.

As it can be derived from their description, strong consistency can only be achieved when using Quorum and All consistency levels. More specifically, strong consistency can be guaranteed as long as equations 11 and 12 hold true. The former ensures that a read opera-

LEVEL	DESCRIPTION
One	Get data from the first node to respond.
Quorum	Wait until majority of replicas respond.
All	Wait for all replicas to respond.

Table 5.1: Cassandra's read consistency levels.

LEVEL	DESCRIPTION
Zero	Return immediately, write value asynchronously.
Any	Write value or hint to at least one node.
One	Write value to log and memtable of at least one node.
Quorum	Write to majority of replicas.
All	Write to all replicas.

Table 5.2: Cassandra's write consistency levels.

tion will always reflect the most recent write, while the latter ensures the consistency of concurrent write operations.

$$\textit{Write replicas} + \textit{Read replicas} > \textit{Replication factor} \tag{11}$$

$$\textit{Write replicas} + \textit{Write replicas} > \textit{Replication factor} \tag{12}$$

Operations that use weaker consistency levels, such as Zero, Any and One, are not guaranteed to read the most recent data. However, this weaker consistency provides certain flexibility for applications that can benefit from better performance and do not have strong consistency needs.

5.2.2.1 *Extending Cassandra's Isolation*

While Cassandra's consistency is tunable, it does not offer a great deal of flexibility when compared to traditional databases and its support for transactions. Cassandra applications could benefit from extended isolation support, which would be specially helpful in the environments in which Cassandra is being used, and remove the burden of additional logic on the application side.

Lock-based approaches, used to implement true serializable transactions, are not desirable due to the distributed and non-blocking nature of Cassandra, since locks would have a huge impact on the performance. But there are other approaches that seem more appropriate,

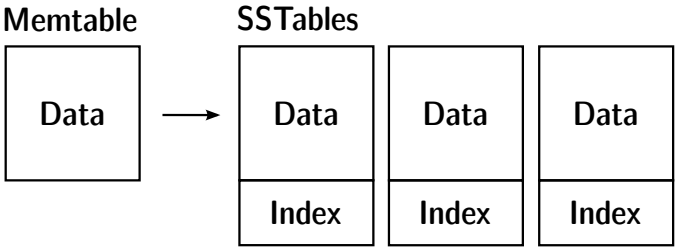


Figure 5.1: Data is persisted in Cassandra by flushing a column family's memtable into an SSTable.

such as multiversion concurrency. Cassandra, unlike other key-value or column stores, does not provide true multiversion capabilities and older versions of the data are not guaranteed to be available in the system, but its timestamps provide a basic notion of versions that can be the basis for multiversion-like capabilities.

Our goal is then to extend Cassandra to support an additional isolation level that will make it possible to provide stronger semantics using a multiversioned approach. In particular, we implement read-only transactions, guaranteeing that reads within a transaction are repeatable and exactly the same. This kind of transactions are specially relevant in the environments in which Cassandra is being adopted, where there's a continuous stream of new data and multiple consumers that sometimes need to operate on a consistent view of the database. Our proposal is similar to Snapshot Isolation in that it guarantees that all reads made in a transaction see the same *snapshot* of the data, but it is not exactly the same since we are not concerned with conflicting write operations. Hence from now on we call this new isolation level Snapshotted Reads.

5.2.3 *Implementing Snapshotted Reads*

Implementing Snapshotted Reads requires multiple changes in different parts of Cassandra: first, the data store, to enable creating and maintaining versioned snapshots of the data, and second, the reading path, in order to read specific versions of the data.

5.2.3.1 *Data Store*

Cassandra nodes handles data for each column family using two structures: memtable and SSTable. A memtable is basically an in-memory write-back cache of data; once full, a memtable is flushed to disk as an SSTable. So, while there is a single active memtable per node and column family, there is usually a larger number of associated SSTables, as shown in Figure 5.1. Also, note that when memtables are persisted to disk as SSTables, an index and a bloom filter are also written along with the data, so as to make queries more efficient.

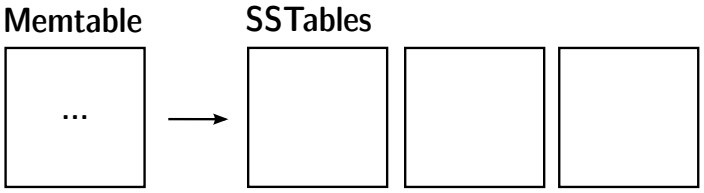


Figure 5.2: State of a column family in a Cassandra node before starting a Snapshotted Read transaction.

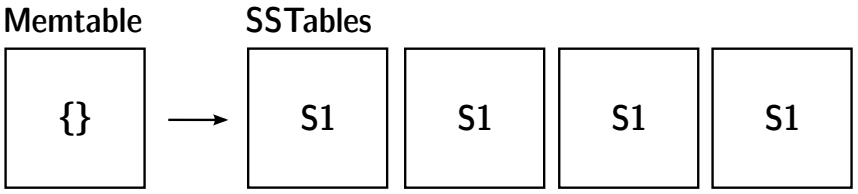


Figure 5.3: State of a column family in a Cassandra node after starting a Snapshotted Read transaction and creating snapshot S1.

Once a memtable is flushed, its data is immutable and can't be changed by applications, so the only way to update a record in Cassandra is by textitappending data with a newer timestamp.

Our implementation of Snapshotted Reads takes advantage of the fact that SSTables are immutable to allow keeping multiple versions of the data and thus providing effective snapshots to different transactions. This mechanism is described in the following figures. Figure 5.2 shows the data for a column family stored in a particular node: there is data in memory as well as in three SSTables. Once we begin a Snapshotted Read transaction, a new snapshot of the data is created by 1) emptying the memtable, flushing its data into a new SSTable, and 2) assigning an identifier to all SSTables, as shown in Figure 5.3.

After the snapshot is created, the transaction will be able to read from it for as long as the transaction lasts, even if other transactions keep writing data to the column family. It is also possible for multiple transactions to keep their own snapshots of the data, as shown in the following figures. Figure 5.4 shows the state of column family when writes occur after a snapshot (S1). Writes continue to operate as expected, eventually creating new SSTables: in this particular example, there is new data in the memtable as well as in two SSTables. If a transaction were to begin a new Snapshotted Read and create new snapshot, the procedure would be the same: flush and assign identifiers to SSTables, as shown in Figure 5.5.

5.2.3.2 *Reading and Compacting*

Reading from a snapshot during a transaction is a matter of selecting data from the appropriate SSTables during the collation process, ignoring SSTables that are not part of the snapshot.

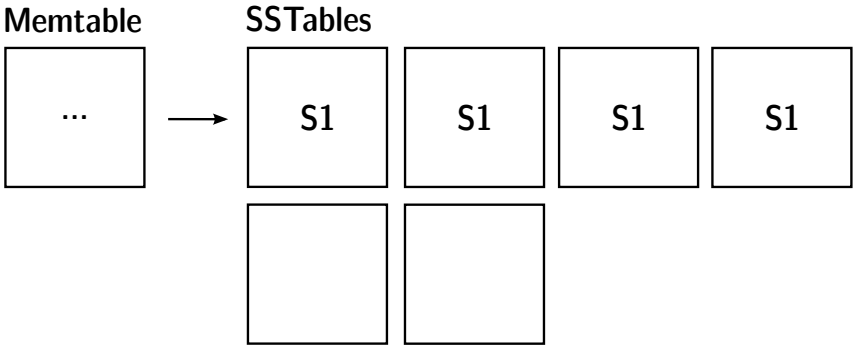


Figure 5.4: State of a snapshotted column family in a Cassandra node after some additional writes.

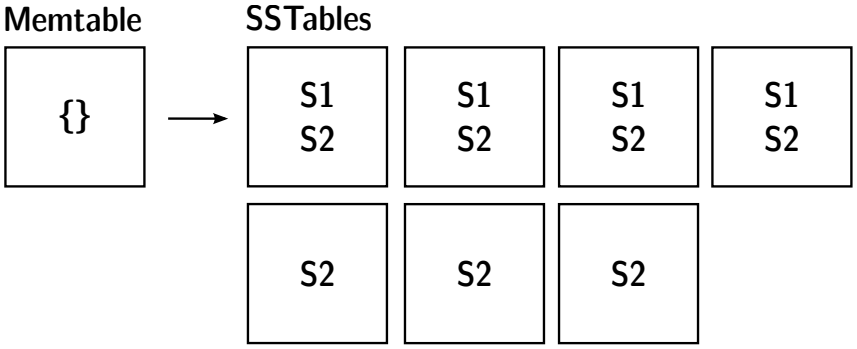


Figure 5.5: State of a column family with two snapshots (S1, S2).

However, multiple records may still be available even within a snapshot, and Cassandra provides mechanisms to address this kind of diverging results, such as read repair. Read repair simply pushes the most recent record to all replicas when multiple records are found during a read operation. Since there is no way to push new data to an older snapshot, read repair is disabled for Snapshotted Read transactions.

Compaction, on the other hand, is a background operation that merges SSTables, combining its columns with the most recent record and removing records that have been overwritten. Compaction removes duplication, freeing up disk space and optimizing the performance of future reads by reducing the number of seeks. The default compaction strategy is based on the size of SSTables, but does not consider snapshots, so it may delete relevant data for our transactions.

We have implemented a new compaction strategy that takes into account snapshots. The main idea behind the new strategy is to compact SSTables only within certain boundaries, namely snapshots. Using this bounded compaction strategy, only SSTables that share exactly the same set of snapshots are considered for compaction. For instance, continuing with the same example in Figure 5.5, compaction can only be applied to the older four SSTables (identified as S1 and S2)

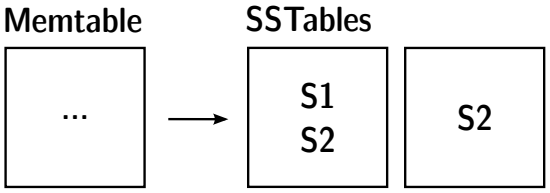


Figure 5.6: State of a column family in a Cassandra node with two snapshots after a bounded compaction.

or the remaining three SSTables (identified as S2). One of the possible outcomes of a major compaction is shown in Figure 5.6.

5.2.3.3 *API Extension*

Finally, in order to support snapshots, some changes have been made to Cassandra's API, including 3 new operations: create snapshot, delete snapshot, and get data from a particular snapshot.

Operations to create or delete a snapshot take 2 arguments: first, the snapshot identifier, and then, optionally, the name of a column family. If no column family is specified, all column families in the current namespace are snapshotted. The operation to retrieve data from a snapshot resembles and works exactly like Cassandra's standard *get* operation with an additional argument to specify the snapshot identifier from which the data is to be retrieved.

5.2.4 *Evaluation*

In this section we include results from three experiments that explore the performance of our implementation of Snapshotted Reads for Cassandra. Experiment 1 shows the overall performance of the system under different loads in order to compare the maximum throughput achievable with each version of Cassandra. In Experiment 2 we compare Cassandra with and without Snapshotted Read support using a synthetic benchmark in order to see what is the impact of keeping snapshots under different workloads trying to achieve maximum throughput. Finally, Experiment 3 studies how does our implementation perform and scale in the presence of multiple snapshots.

5.2.4.1 *Environment*

The following experiments have been executed on a Cassandra cluster consisting of 20 Quad-Core 2.13 GHz Intel Xeon machines with a single SATA disk and 12 GB of memory, connected with a gigabit ethernet network. The version of Cassandra used for all the experiments is 1.1.6.

In these experiments we run the synthetic workloads provided by the Yahoo! Cloud Serving Benchmark (YCSB) tool [23]. The workloads are defined as follows:

A: UPDATE HEAVY. Read/update ratio: 50%/50%. Application example: session store recording recent actions.

B: READ MOSTLY. Read/update ratio: 95%/5%. Application example: photo tagging; add a tag is an update, but most operations are to read tags.

C: READ, MODIFY, WRITE. Read/read-modify-write ratio: 50%/50%. Application example: user database, where user records are read and modified by the user or to record user activity.

D: READ ONLY. Read/update ratio: 100%/0%. Application example: user profile cache, where profiles are constructed elsewhere (e.g. Hadoop).

E: READ LATEST. Read/insert ratio: 95%/5%. Application example: user status updates, people want to read the latest.

The execution of each workload begins with the same initial dataset, which consists of 380,000,000 records (approximately 400 GB in total) stored across the 20 nodes of the cluster with a single replica, meaning each node stores approximately 20 GB of data, and thus exceeds the capacity of the memory. During each execution, a total of 15,000,000 read and/or write operations, depending on the workload, are executed from 5 clients on different nodes. Cassandra nodes are configured to run the default configuration for this system, consisting of 16 threads for read operations and 32 threads for writes.

The following tables and figures show the results of running the workloads with two different versions of Cassandra: the original version and our version of Cassandra with Snapshotted Read support. Note that for our version of Cassandra we also compare regular reads, which are equivalent to reading in the original Cassandra, regular reads in the presence of a snapshot, and finally snapshotted reads, which get data from one particular snapshot.

5.2.4.2 *Experiment 1: Throughput*

In this experiment we execute two workloads with different configurations in order to explore how does Cassandra perform reads under different loads (which are specified to the YCSB client as target throughputs). We first study the workload D, which only performs read operations, since our changes to Cassandra are focused on the read path. We then execute workload A in order to validate the results under update-intensive workloads.

Tables and figures in this section show the four different kinds of ways to read data from Cassandra that we compare in this experiment: the first one reading from the Original Cassandra, and the remaining three reading from Cassandra with Snapshotted Read support. In particular, for Cassandra with Snapshotted Read Support we evaluate performing regular reads (S/R), performing regular reads in the presence of a snapshot (S/RwS), and performing Snapshotted Reads (S/SR). The measured results are the average and corresponding standard deviation after running 5 executions of each configuration.

Table 5.3 and Figure 5.7 show the results of running workload D. As it can be observed, latency is similar under all configurations for each target throughput, and the same pattern can be observed in all executions: on the one hand the performance of regular reads is similar, independently of the version of Cassandra and the presence of the snapshot, and on the other hand reading from the snapshot is slightly slower with a slowdown around 10%.

Operations/s	Original	S/R	S/RwS	S/SR
1000	6.09	6.15	6.18	6.54
2000	7.44	7.46	7.64	7.96
3000	10.15	10.44	10.51	11.42
4000	13.18	13.33	13.45	14.06
5000	18.47	18.46	18.60	19.39

Table 5.3: Average read latency (ms) of Workload D using Original Cassandra and Cassandra with Snapshotted Read support (S/R, S/RwS, SR)

Figure 5.7 also shows the standard deviation of the executions, and the real throughput achieved with each target (shown as a black line). As it can be seen in this workload, the observed throughput grows linearly with the target throughput until its optimal level, which is approximately 4400 operations per second when running the Original Cassandra. After reaching the maximum throughput, latency simply degrades without any benefit in terms of throughput.

Similarly, Table 5.4 and Figure 5.8 show the results of running workload A (50% read, 50% update) under different loads. The main difference in this workload compared to workload D (100% read) is the performance of reading from the snapshot, which is slightly faster. If more than one record is found when reading a key, Cassandra will merge the multiple records and select the right one. However, in this particular experiment, the snapshot is created under perfect conditions and SSTables are fully compacted. So, while regular reads may degrade slightly over time as new SSTables are created by updates

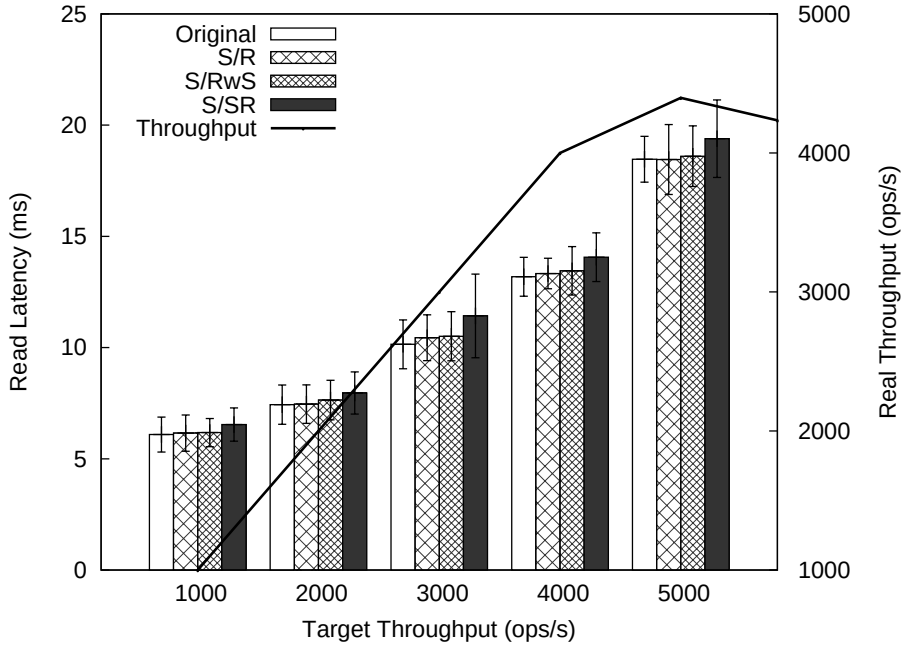


Figure 5.7: Average read latency and observed throughput for varying targets of operations per second on Workload D

and not yet compacted, the latency of snapshotted reads remains mostly the same since snapshots are not updated.

Operations/s	Original	S/R	S/RwS	S/SR
4000	8.25	8.47	8.44	7.71
5000	9.25	9.25	9.22	8.68
6000	11.32	11.69	11.57	10.38
7000	14.19	14.42	14.30	13.76
8000	15.36	15.77	15.45	14.78
9000	18.59	18.73	18.97	18.17

Table 5.4: Average read latency (ms) of Workload A using Original Cassandra and Cassandra with Snapshotted Read support (S/R, S/RwS, SR)

Therefore, the differences in the performance when reading from a snapshot depending on the kind of workload can be explained by how Cassandra handles read and write operations, and the strategy used to compact SSTables. As described in Section 2.5.1, Cassandra first writes data to a memtable, and once it is full it is flushed to disk as a new SSTable, which eventually will be compacted with other SSTables. Compaction is not only important to reclaim unused space, but also to limit the number of SSTables that must be checked when performing a read operation and thus its performance. Our version

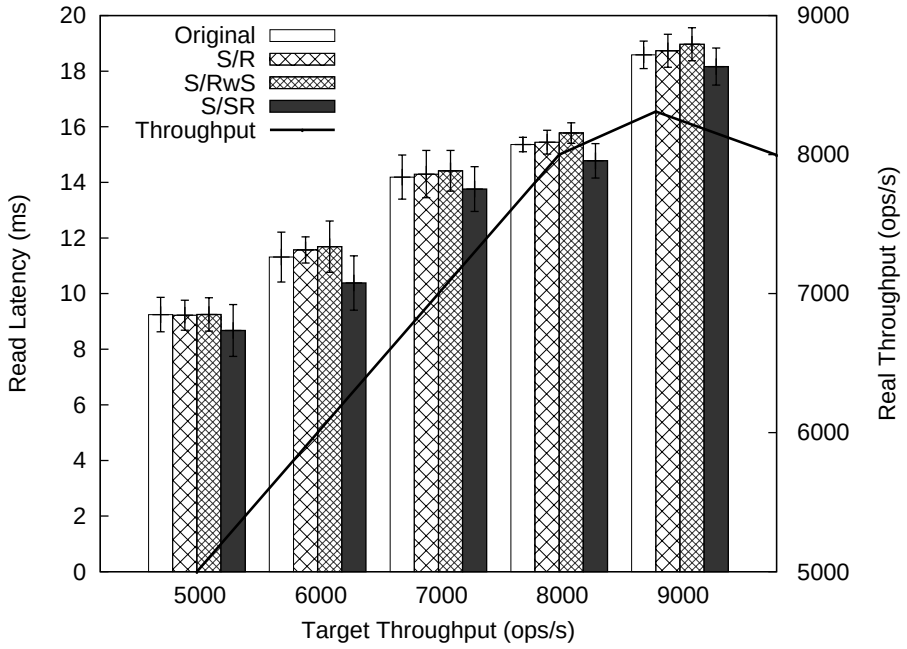


Figure 5.8: Average read latency and observed throughput for varying targets of operations per second on Workload A

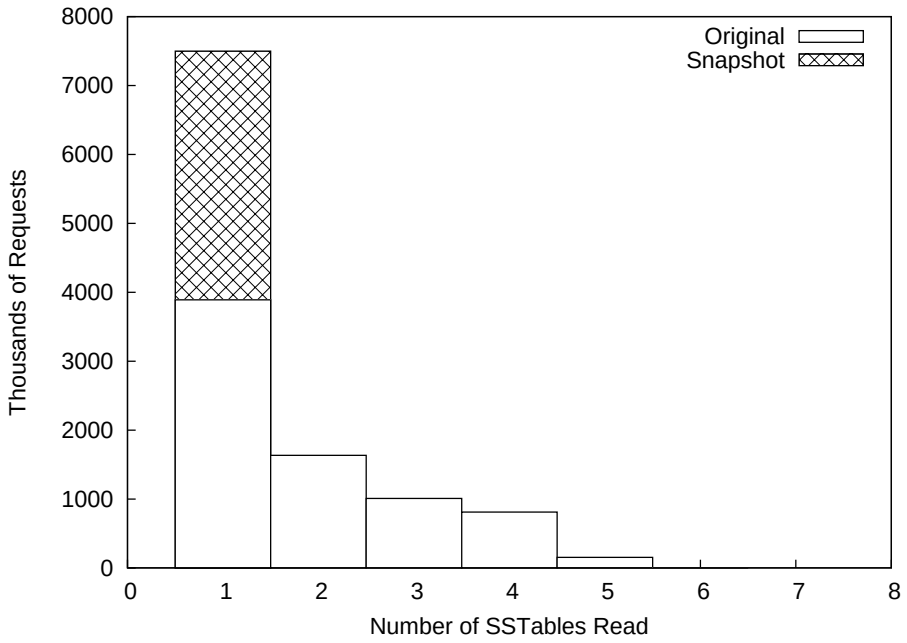


Figure 5.9: Distribution of number of SSTables read for each read operation on workload A when performing regular and snapshotted reads

of Cassandra with Snapshotted Read support uses a custom compaction strategy, as described in Section 5.2.3.2. While our bounded compaction strategy is necessary to keep data from snapshots, it also makes compaction less likely since it will not allow compaction between snapshot boundaries.

The consequences of this behaviour for Cassandra with Snapshotted Read support are twofold: first, as observed, reading from a snapshot may be faster on workloads in which data is mostly updated (and snapshots eventually consolidated), and second, it may make regular reads slower when snapshots are present, due to the increased amount of SSTables caused by our compaction strategy.

Figure 5.9 shows the distribution of how many SSTables must be checked during read queries in workload A. As it can be observed, there is a significant difference between reading from a snapshot and performing a regular read. While snapshotted reads always get the data from a single SSTable, regular reads require checking two SSTables or more at least half of the time. Again, it should be noted that there is nothing that makes snapshot intrinsically less prone to spreading reads to multiple SSTables, it is simply their longer-term nature that helps consolidate and compact snapshots. Regular reads, on the other hand, need to deal with the latest updates, so they are more likely to be spread across multiple SSTables.

In order to compare how does the number of SSTables impact our version of Cassandra, we also executed the update-intensive workload A, and then increased the frequency at which the number of SSTables are generated, thus increasing the number of SSTables. As it can be observed in Figure 5.10, while performing regular reads with Original Cassandra becomes slower when we force a larger number of SSTables, reading from a snapshot remains mostly unchanged since it simply reads from the same subset of SSTables all the time.

5.2.4.3 *Experiment 2: Read Latency*

In this experiment we compare the latency of reading operations under different workloads in order to find out what's the impact of supporting snapshotted reads as well as what's the performance of reading from a snapshot compared to a regular read under a wider variety of scenarios.

This experiment compares two different kinds of ways to read: the first one reading from the Original Cassandra, and the second one reading from a snapshot on our version of Cassandra with Snapshotted Read support. We omit here the results of regular reads on our version of Cassandra since they are similar to Original Cassandra. All workloads are executed 5 times, and the YCSB client is configured to run with the maximum number of operations per second.

These workloads show different behaviours. One the one hand, in *read-modify* workloads, including A and B, data is updated but the

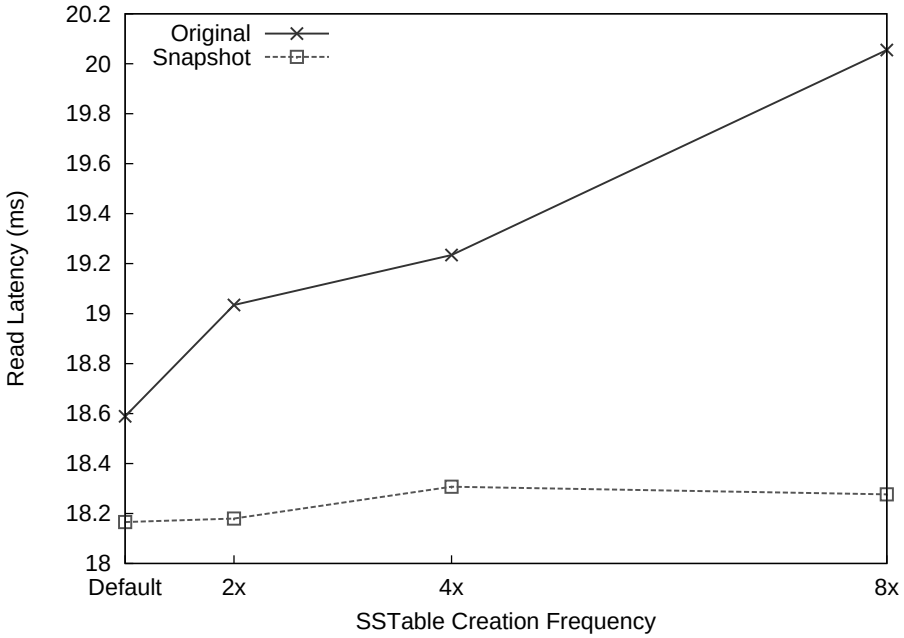


Figure 5.10: Average read latency on Workload A, performing regular and snapshotted reads, and varying the frequency at which SSTables are created relative to to the default configuration

size of the data set (number of keys) remains the same. On the other hand, in *write-once* workloads, which include D and E (read-only and read-insert respectively), each record is only written once and it's not modified afterwards. Finally, workload C can be thought of as a special case since half of the operations are composed and perform a read followed by a write operation.

As shown in Table 5.5 and Figure 5.11, the latency of reading from a single snapshot is similar to performing regular reads. Generally speaking, reading from the original Cassandra is slightly faster, and the slower reads are from our version of Cassandra performing snapshotted reads on read-intensive workloads.

Table 5.5: Average read latency (ms) using Original Cassandra and Cassandra with Snapshotted Read support (Regular, Snapshot)

Workload	Original	Snapshot
A	18.59	18.17
B	18.66	18.75
C	19.08	20.04
D	18.47	19.39
E	12.07	12.73

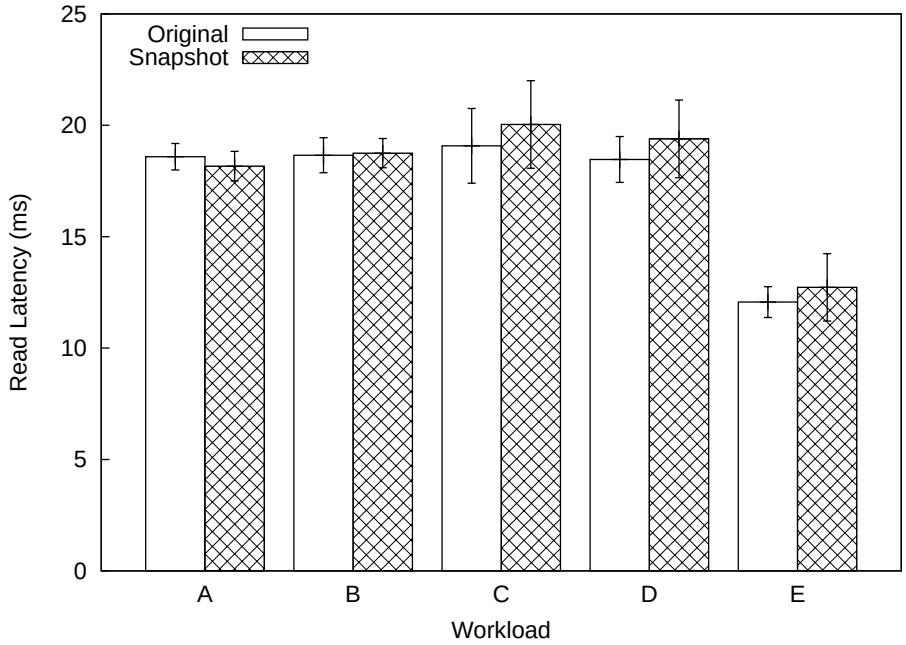


Figure 5.11: Average read latency for each workload, comparing regular reads to reading from a snapshot

As it can be observed, *read-modify* workloads (A, B) are the workloads that remain closer to each other, independently of the kind of read we are performing. Workload A remains faster when reading from a snapshot than when reading regularly, while snapshotted read under workload B is slightly slower since the amount of updates is relatively small and thus regular reads almost always involve a single SSTable. On the other hand, both *write-once* workloads (D, E) display a much more noticeable difference between the two kinds of reads since data is only written once and so each operation reads from exactly one SSTable.

5.2.4.4 *Experiment 3: Increasing the Number of Snapshots*

While the previous experiments discuss the performance of different kinds of reads with a single fully-compacted snapshot, in this experiment we evaluate the evolution of the performance under a more realistic scenario in which multiple snapshots are created and read during its execution.

In order to test multiple snapshots and compare the results of previous experiments, we execute workload A ten times consecutively one after another. In particular, we execute 3 different versions in this experiment: first the original Cassandra performing regular reads, and then our version of Cassandra, either creating a single snapshot at the beginning (S/1), or creating a new snapshot for each iteration (S/N). Since workload A involves at least 50% of update operations,

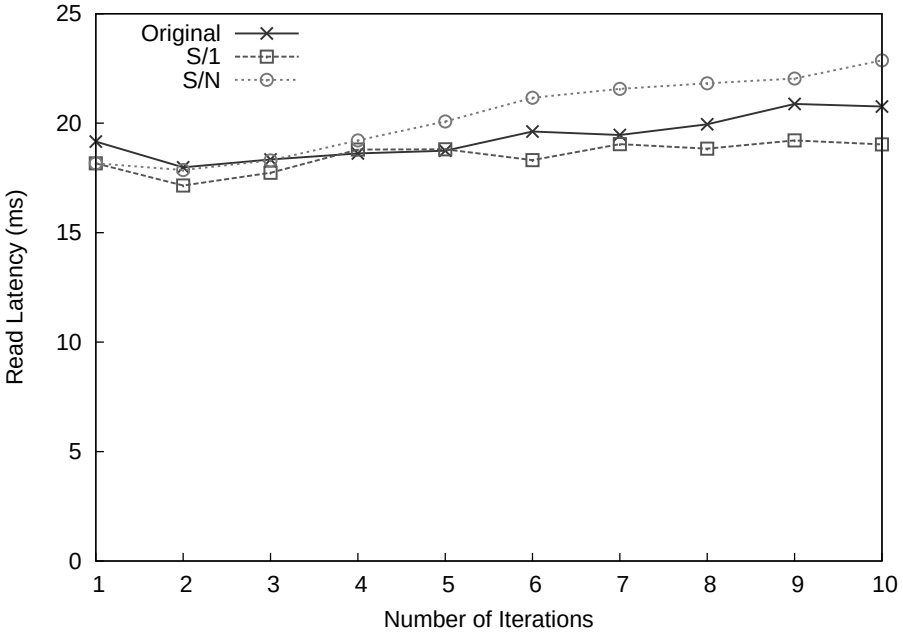


Figure 5.12: Evolution of average read latency for 10 consecutive executions of Workload A

we ensure an increasing number of SSTables as new snapshots are created.

As shown in Figure 5.12, after the first few executions, the performance of read operations degrades slightly over time as new consecutive executions of workload A are completed, independently of the version of Cassandra we are running. Regular reads with Original Cassandra and snapshotted reads with our version of Cassandra and a single snapshot both become more stable after a few iterations and do not change too much afterwards. However, as it could be expected, departing from the initial scenario with fully compacted SSTables and keeping multiple snapshots becomes noticeably slower over time as shown in the Figure. When creating a new snapshot for each iteration (S/N), read latency goes from 18.17 ms during the first iteration to 22.87 after all iterations with 10 snapshots.

The varying performance can also be explained in terms of how the data is read and stored as SSTables. For instance, while with the original version of Cassandra there are 193 SSTables in the cluster after all executions, with our version of Cassandra creating a new snapshot for each iteration (S/N) there are as many as 532 SSTables. Figure 5.13 also shows the evolution of the distribution of SSTables read for each operation. As expected, at the beginning when there is only one snapshot and the data is still well compacted, all operations only read from a single SSTable. However, as soon as we introduce more snapshots (3 and 5 as shown in the Figure), the number of seeks to SSTables for each read operation increases as well, thus making read operations slower.

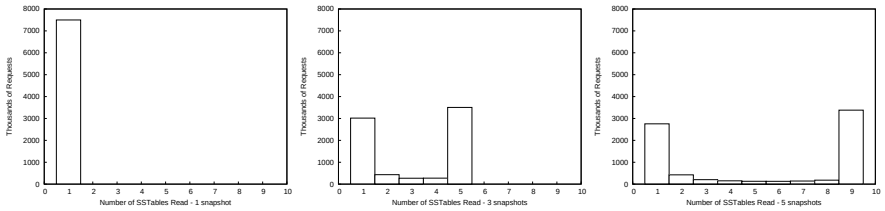


Figure 5.13: Distribution of number of SSTables read for each read operation on workload A with multiple snapshots

5.2.5 *Related Work*

There have been many efforts to implement features usually available in relational databases on top of distributed data stores [3, 10, 36], and, as other have pointed out [55, 28], this further proves that some of their functionality is converging. Isolation and transactional support for distributed data stores is also a widely studied topic, and there has been some related work done, including support for lock-free transactions [44] and snapshot isolation [68] for distributed databases.

There has also been work more focused on stronger semantics for distributed key-value data stores. Google's Percolator [56] implements snapshot isolation semantics by extending Bigtable with multi-version timestamp ordering using a two-phase commit, while Spanner [28] and Megastore [13] also provide additional transactional support for Bigtable. In [92] and [93] the authors also implement snapshot isolation for HBase, allowing multi-row distributed transactions for this column-oriented database. While the former approach uses additional meta-data on top of standard HBase, the latter introduces a more advanced client to support snapshot isolation transactions.

There has not been much work done in the space of isolation for Cassandra in particular since improving it is orthogonal to its design, and other than the configurable consistency levels, there is basically no support for transactions. Cassandra currently only provides support to create *backup* snapshots, which are only meant to be used as a way to backing up and restoring data on disk. So, unlike our proposal, with *backup* snapshots it is only possible to read from one of these snapshots at at time and reading from a different snapshot involves reloading the database.

5.3 ADAPTIVE MAPREDUCE SCHEDULING IN SHARED ENVIRONMENTS

In this section we present a MapReduce task scheduler for shared environments in which MapReduce is executed along with other resource-consuming workloads, such as transactional applications. All workloads may potentially share the same data store, some of them

consuming data for analytics purposes while others acting as data generators. This kind of scenario is becoming increasingly important in data centers where improved resource utilization can be achieved through workload consolidation, and is specially challenging due to the interaction between workloads of different nature that compete for limited resources. The proposed scheduler aims to improve resource utilization across machines while observing completion time goals. Unlike other MapReduce schedulers, our approach also takes into account the resource demands for non-MapReduce workloads, and assumes that the amount of resources made available to the MapReduce applications is variable over time. As shown in our experiments, our proposal improves the management of MapReduce jobs in the presence of variable resource availability, increasing the accuracy of the estimations made by the scheduler, thus improving completion time goals without an impact on the fairness of the scheduler.

5.3.1 *Introduction*

In recent years, the industry and research community have witnessed an extraordinary growth in research and development of data-related technologies. In addition to distributed, large-scale data processing workloads such as MapReduce [24], other distributed systems have been introduced to deal with the management of huge amounts of data [19] [25] providing at the same time support for both data-analytics and transactional workloads.

Instead of running these services in completely dedicated environments, which may lead to underutilized resources, it is becoming more common to multiplex different and complementary workloads in the same machines. This is turning clusters and data centers into shared environments in which each one of the machines may be running different applications simultaneously at any point in time: from database servers to MapReduce jobs to other kinds of applications [14]. This constant change is challenging since it introduces higher variability and thus makes performance of these systems less predictable.

In particular, in this section we consider an environment in which data analytics jobs, such as MapReduce applications, are collocated with transactional workloads. In this scenario, deep coordination between management components is critical, and single applications can not be considered in isolation but in the full context of mixed workloads in which they are deployed. Integrated management of resources in presence of MapReduce and transactional applications is challenging since the demand for transactional workloads is known to be bursty and varying over time, while MapReduce schedulers usually expect that available resources are unaltered over time. Transactional workloads are usually of higher priority than analytics jobs

because they are directly linked to the QoS perceived by the users. As such, in our approach transactional workloads are considered as critical and we assume that only resources not needed for transactional applications can be committed to MapReduce jobs.

In this work we present a novel scheduler, the Reverse-Adaptive Scheduler, that allows the integrated management of data processing frameworks such as MapReduce along with other kinds of workloads that can be used for both, transactional and analytics workloads. The scheduler expects that each job is associated a completion time goal that is provided by users at job submission time. These goals are treated as soft deadlines as opposed to the strict deadlines familiar in real-time environments: they simply guide workload management. We also assume that the changes in workload intensity over time for transactional workloads can be well characterised, as has been previously stated in the literature [53].

Existing previous work on MapReduce scheduling involved estimating the resources that needed to be allocated to each job in order to meet its completion goals [59, 63, 82]. This naive estimation worked fine under the assumption that the total amount of resources remained stable over time. However, in a scenario with consolidated workloads we are targeting a more dynamic environment in which resources are shared with other frameworks and availability changes depending on external and a priori unknown factors. The scheduler proposed here proactively deals with dynamic resource availability while still being guided by completion time goals.

While resource management has been widely studied in MapReduce environments, to our knowledge no previous work has focused on shared scenarios with transactional workloads.

The remaining sections are organized as follows. We first present a motivating example to illustrate the problem that the proposed scheduler aims to address in Section 5.3.2. After that, we provide an overview of the problem in Section 5.3.3, and then describe our scheduler in Section 5.3.4. An evaluation of our proposal is studied in Section 5.3.5. Finally, Section 5.3.6 discusses the related work.

5.3.2 *Motivating example*

Consider a system running two major distributed frameworks: a MapReduce deployment used to run background jobs, and a distributed data-store that handles transactional operations and serves data to a front-end. Both workloads share the same machines, but since the usage of the front-end changes significantly over time depending on external the activity of external entities, so does the availability of resources left for the MapReduce jobs. Notice that the demand of resources over time for the front-end activities is supposed to be well

characterized [53], and therefore it can be assumed to be known in advance in the form of a given function $f(t)$.

In the proposed system, the MapReduce workload consists of 3 identical jobs: J1, J2, and J3. All jobs are submitted at time 0, but have different deadlines: D1 (6.5h), D2 (15h), and D3 (23.1h). Co-located with the MapReduce jobs, we have a front-end driven transactional workload that consumes available resources over time. The amount of resources committed to the critical transactional workload is defined by the function $f(t)$.

Figure 5.14 shows the expected outcome of an execution using a MapReduce scheduler that is not aware of the dynamic availability of resources and thus assumes resources remain stable over time. Figure 5.15 shows the behaviour of a scheduler aware of changes in availability and capable of leveraging the characteristics of other workloads to scheduler MapReduce jobs. In both Figures, the solid thick line represents $f(t)$. Resources allocated to the transactional workload are shown as the white region on top of $f(t)$, while resources allocated to the MapReduce workload are shown below $f(t)$, being each job represented by a different pattern. X-axis shows time, while Y-axis represents compute nodes allocated to the workloads.

Figure 5.14 represents the expected behavior of a scheduler that is not aware of the presence of other workloads. As it is not able to predict a future reduction in available resources, it is not able to cope with dynamic availability and misses the deadline of the first two jobs because it unnecessarily assigns tasks from all jobs (e.g. from time 0 to 5, and from time 7 to 11 approximately). On the other hand, Figure 5.15 shows the behaviour of the scheduler presented in this section, the Reverse-Adaptive Scheduler, which distributes nodes across jobs considering future availability of resources. From time 0 to D1, most of the tasktrackers are assigned tasks from J1, and the remaining to J2 since it also needs those resources to reach its goal on time. From time D1 to D2, most of the resources go to J2 in order to meet a tight goal. However, as soon as J2 is estimated to reach its deadline, a few tasks from J3 are assigned as well starting around time 4. Finally, from time D2 until the end only tasks from J3 remain to be executed.

5.3.3 *Problem Statement*

We are given a cluster of machines, formed by a set of nodes $\mathcal{N} = \{1, \ldots, N\}$ in which we need to run different workloads. We use n to index the set of nodes. We are also given a set of MapReduce jobs $\mathcal{J} = \{1, \ldots, J\}$, that has to be run in $\mathcal{N}$. We use j to index the set of MapReduce jobs.

Each node n hosts two main processes: a MapReduce slave and a non-MapReduce process that represents another kind workload. While MapReduce usually consists of a tasktracker and a datanode

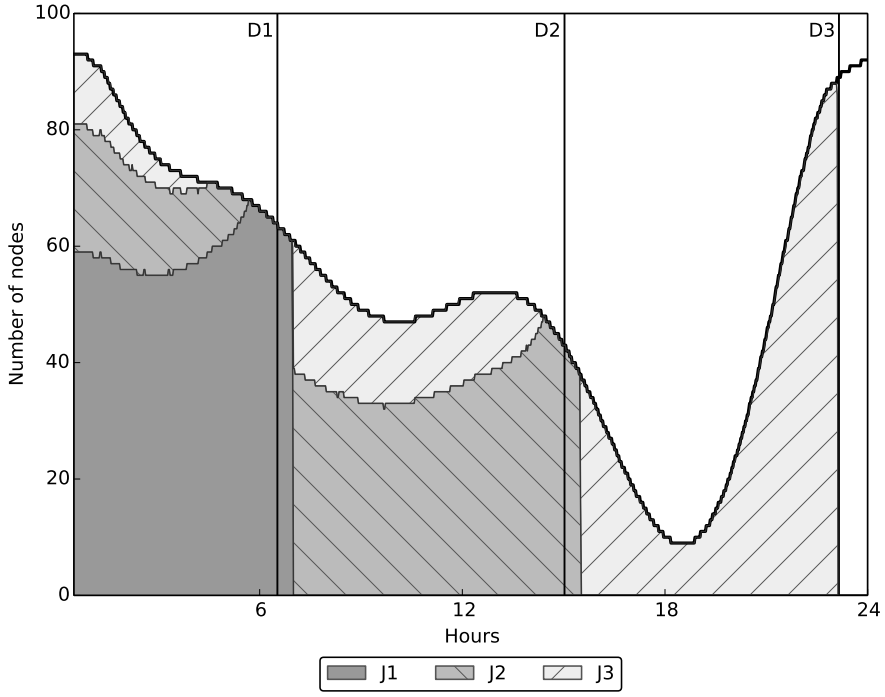


Figure 5.14: Distribution of assigned resources over time running the sample workload using a scheduler without dynamic resource availability awareness

in Hadoop terminology, we summarize both of them for simplicity and refer to them as the *tasktracker* process hereafter. Similarly, The non-MapReduce process could represent any kind of workload but we identify it as *data-store* in this section.

We refer to the set of MapReduce processes, or tasktrackers, as $\mathcal{TT} = \{1, \ldots, N\}$ and the set of data-store processes committed to the front-end activity as $\mathcal{DS} = \{1, \ldots, N\}$, and we use tt and ds respectively to index these sets.

With each node n we associate a series of resources, $\mathcal{R} = \{1, \ldots, R\}$. Each resource of node n has an associated capacity $\Omega_{n,1}, \ldots, \Omega_{n,r}$, which is shared between the capacity allocated to the tasktracker and to the data-store so that $\Omega_{n,1} = (\Omega_{tt,1} + \Omega_{ds,1}), \ldots, \Omega_{n,r} = (\Omega_{tt,r} + \Omega_{ds,r})$.

The usage of each data-store ds, and thus each of its resources $\Omega_{ds,1}, \ldots, \Omega_{ds,r}$, changes over time depending on the demand imposed by its users, defined by a function $f(t)$. In turn, since the capacity of each node remains the same, the available resources for each tasktracker tt, $\Omega_{tt,1}, \ldots, \Omega_{tt,r}$, also changes to adapt to the remaining capacity left by the data-store.

A MapReduce job (j) is composed of a set of tasks, already known at submission time, that can be divided into map tasks and reduce tasks. Each tasktracker tt provides to the cluster a set of job-slots in which tasks can run. Each job-slot is specific for a particular job, and

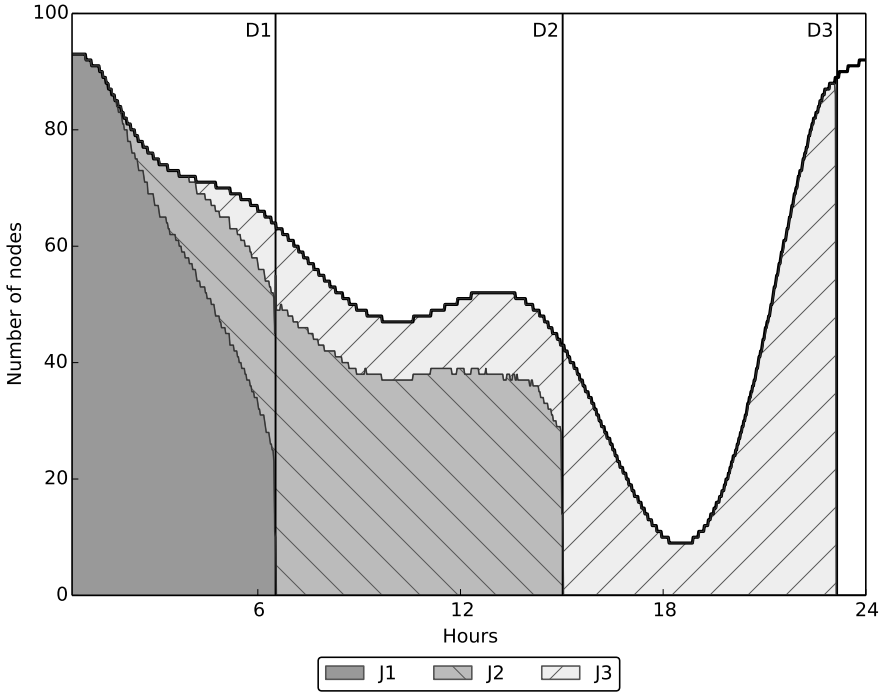


Figure 5.15: Distribution of assigned resources over time running the sample workload using the Reverse-Adaptive Scheduler

the scheduler will be responsible for deciding the number of job-slots to create on each tasktracker for each job in the system.

Each job j can be associated with a completion time goal, T^j_{goal}, the time at which the job should be completed. When no completion time goal is provided, the assumption is that the job needs to be completed at the earliest possible time.

Additionally, with each job we associate a resource consumption profile. The resource usage profile for a job j consists of a set of average resource demands $\mathcal{D}_j = \{\Gamma_{j,1}, \ldots, \Gamma_{j,r}\}$. Each resource demand consists of a tuple of values. That is, there is one value associated for each task type and phase (map, reduce in shuffle phase, and reduce in reduce phase, including the final sort).

We use symbol P to denote a placement matrix with the assignment of tasks to tasktrackers, where cell $P_{j,tt}$ represents the number of tasks of job j placed on tasktracker tt. For simplicity, we analogously define P^M and P^R, as the placement matrix of Map and Reduce tasks. Notice that $P = P^M + P^R$. Recall that each task running in a tasktracker requires a corresponding slot to be created before the task execution begins, so hereafter we assume that placing a task in a tasktracker implies the creation of an execution slot in that tasktracker.

5.3.4 *Reverse-Adaptive Scheduler*

The driving principles of the scheduler are resource availability awareness and continuous job performance management. The former is used to decide task placement on tasktrackers over time, while the latter is used to estimate the number of tasks to be run in parallel for each job in order to meet performance objectives, expressed in the form of completion time goals. Job performance management has been extensively evaluated and validated in our previous work, presented as the Adaptive Scheduler [59] [63]. This section extends the resource availability awareness of the scheduler when the MapReduce jobs are collocated with other time-varying workloads.

One key element of this proposal is the variable S_{fit}, which is an estimator of the minimal number of tasks that should be allocated in parallel to a MapReduce job to keep its chances to reach its deadline, assuming that the available resources will change over time as predicted by $f(t)$. Notice that the novelty of this estimator is the fact that it also considers the variable demand of resources introduced by other external workloads. Thus, the main components of the Reverse-Adaptive Scheduler, as described in the following sections, are:

- S_{fit} estimator. Described in Section 5.3.4.2.
- Utility function that leverages S_{fit} used as a per-job performance model. Described in Section 5.3.4.3.
- Placement algorithm that leverages the previous two components. Described in Section 5.3.4.4.

5.3.4.1 *Intuition*

The intuition behind the reverse scheduling approach is that it divides time into stationary periods, in which no job completions are expected. One period ends and starts in instants in which a job completion time goal is expected. When a job is expected to complete at the end of a period, the scheduler calculates the amount of resource to be allocated during the period for the job to make its completion goal. If the available resources are not enough, the amount of pending work is pushed back to the immediately preceding period. Notice that the amount of the available resources for the period is determined by the function $f(t)$, that estimates the resources that will have to be committed to the non-MapReduce workloads. When more than one job co-exists in the same period, they compete for the available resources, and they are allocated following a fairness criteria that will try to make all jobs obtain the same utility from the decided schedule.

For the sake of clarity, Figure 5.16 retakes the example presented in Section 5.3.2 and shows how the placement decision is made step by step. Starting at the desired completion time, which is represented

by the deadline of the last job, we assign as many tasks as possible from the jobs that are supposed to be running within that timeframe, compressed between that deadline and the previous one. In this case only J3 is running and we are able to assign most of its tasks, as shown in Figure 5.16a. Next we estimate the timeframe between time 17 and 38 as shown in Figure 5.16b, in which we would like to run all the tasks from J2 and the remaining ones from J3. The scheduler is able to run the remaining tasks from J3, but since there are not enough resources to run all the tasks from J2, the remaining ones are carried to the last timeframe. Similarly, in the final step of the estimation as shown in Figure 5.16c, the scheduler evaluates the timeframe between 1 and 17, in which it is supposed to execute J1 and the remaining tasks from J2.

Once the estimation of expected availability is completed, the scheduler is aware of all the steps needed to reach its desired state from the current state, and therefore proceeds to create the next placement of jobs that will satisfy its final goal.

5.3.4.2 *Estimation of the resources to allocate to each job*

We consider a scenario in which jobs are dynamically submitted by users. Each submission includes both the job's completion time goal (if one is provided) and its resource consumption profile. This information is provided via the job configuration file. The scheduler maintains a list of active jobs and a list of tasktrackers. For each active job it stores a descriptor that contains the information provided when the job was submitted, in addition to state information such as number of pending tasks. For each tasktracker tt, the scheduler also knows its resource capacity at any point in time, $\Omega_{tt,1}, \ldots, \Omega_{tt,r}$ since it can be derived from a function that describes the transactional workload pattern, $f(t)$.

For any job j in the system, let s^j_{pend} be the number of map tasks pending of execution. The scheduler estimates the minimum number of *map* tasks that should be allocated concurrently during the next placement cycle, s^j_{fit}, by *reversing* the expected execution assuming all jobs meet their completion time goal T^j_{goal}, and relying on the observed task length (μ^j) and the availability of resources over time (Ω_{tt}).

Algorithm 2 shows how this estimation takes place. We first start assuming that for each job j, s^j_{fit} equals the number of pending tasks s^j_{pend} (lines 1-3), and then proceed to subtract as many tasks as possible beginning from the job with the last deadline to the job with the earliest deadline (lines 5-8), and as long as they *fit* within the available amount of resources (lines 9-17). The algorithm uses the $fit()$ function, which given a job j and two points in time a and b returns the amount of tasks from job j that can be assigned between time a

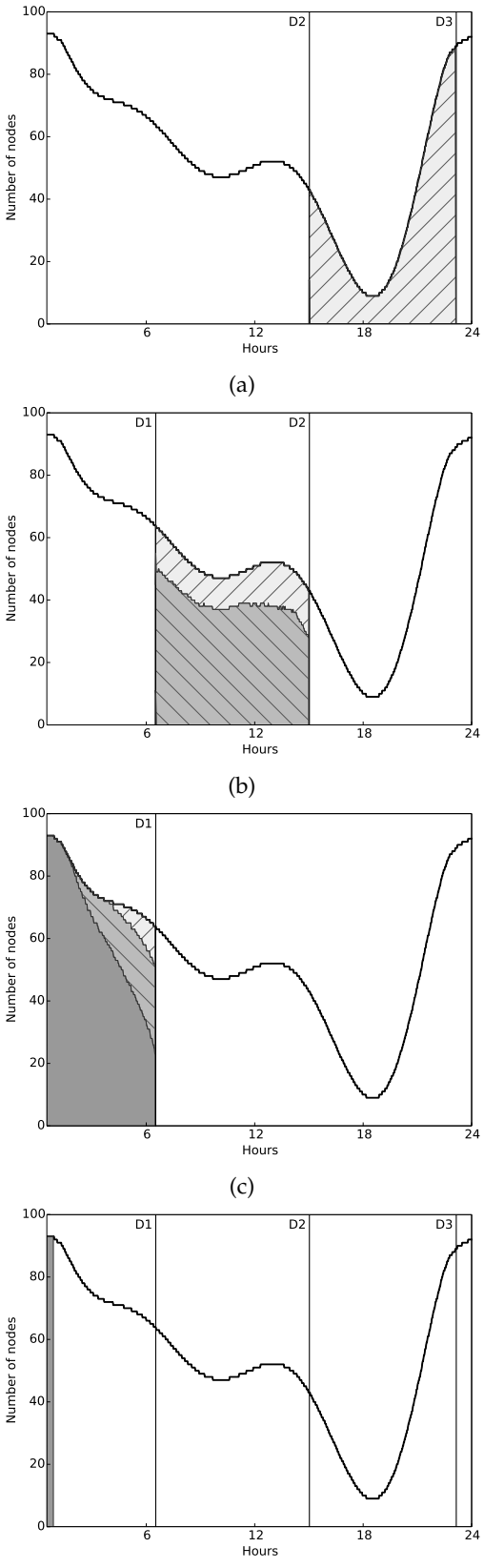


Figure 5.16: Step by step estimation with the Reverse-Adaptive Scheduler from (a) to (c), and placement decision (d)

Algorithm 2 Reverse fitting algorithm to estimate s_{fit}

Inputs J: List of Jobs in the system; s^j_{pend}: Number of pending map tasks for each job; Γ_j and Ω_{tt}: Resource demand and capacity for each job and tasktracker correspondingly, as used by the auxiliary function fit

1: **for** j in J **do**
2: $s^j_{fit} = s^j_{pend}$
3: **end for**
4: $P = []$
5: Sort J by completion time goal
6: **for** j in J **do**
7: $a = T^{next(J)}_{goal}$ // deadline for the next job in J
8: $b = T^j_{goal}$ // deadline for j
9: **for** p in P **do**
10: **if** $s^p_{fit} > 0$ **then**
11: $s^p_{fit} = s^p_{fit} - fit(p, a, b)$
12: **end if**
13: **end for**
14: **if** $s^j_{fit} > 0$ **then**
15: $s^j_{fit} = s^j_{fit} - fit(j, a, b)$
16: **end if**
17: Add j to P
18: **end for**
19: **return** s^j_{fit} for each job in J

and b, taking into consideration the profile and resource requirements of said job. Notice also how on every iteration we try to *fit* tasks between the two last deadlines (lines 7-8), and try to assign tasks from jobs with the latest deadlines first as long as they still have remaining tasks left (lines 9-13).

In addition to the main estimator s^j_{fit}, which estimates the minimum number of tasks to be allocated for each job during the next placement cycle, we also calculate the average number of tasks that should be allocated over time considering a fixed availability of resources equal to the average amount of resources from current time to its deadline, s^j_{req}. The latter is used to assign remaining the resources left after allocating the minimum number of tasks with the former, if any.

5.3.4.3 *Performance Model*

To measure the performance of a job given a placement matrix, we define a utility function that combines the number of map and reduce slots allocated to the job with its completion time goal and job characteristics. Below we provide a description of this function.

Given placement matrices P^M and P^R, we can define the number of map and reduce slots allocated to a job j as $s^j_{alloc} = \sum_{tt \in \mathcal{TT}} P^M_{j,tt}$ and $r^j_{alloc} = \sum_{tt \in \mathcal{TT}} P^R_{j,tt}$ correspondingly.

Based on these parameters and the previous definitions of s^j_{pend} and r^j_{pend}, we define the utility of a job j given a placement P as:

$$u_j(P) = u^M_j(P^M) + u^R_j(P^R), \quad \text{where } P = P^M + P^R \tag{13}$$

and where u^M_j is a utility function that denotes increasing satisfaction of a job given a placement of map tasks, and u^R_j is a utility function that shows satisfaction of a job given a placement of reduce tasks. The definition of both is as follows:

$$u^M_j(P^M) = \begin{cases} \frac{log(s^j_{alloc})}{log(s^j_{fit})} - 1 & s^j_{alloc} < s^j_{fit} \\ \frac{s^j_{alloc} - s^j_{fit}}{2 \times (s^j_{req} - s^j_{fit})} & s^j_{fit} < s^j_{alloc} < s^j_{req} \\ \frac{s^j_{alloc} - s^j_{req}}{2 \times (s^j_{pend} - s^j_{req})} + \frac{1}{2} & s^j_{fit} < s^j_{req} < s^j_{alloc} \\ \frac{s^j_{alloc} - s^j_{fit}}{s^j_{pend} - s^j_{fit}} & s^j_{req} \leq s^j_{fit} < s^j_{alloc} \end{cases} \tag{14}$$

$$u^R_j(P^R) = \frac{log(r^j_{alloc})}{log(r^j_{pend})} - 1 \tag{15}$$

Notice that in practice a job will never get more tasks allocated to it than it has remaining: to reflect this in theory we cap the utility at $u_j(P) = 1$ for those cases.

The definition of u differentiates between three cases: (1) the satisfaction of the job grows logarithmically from $-\infty$ to 0 if the job has fewer map slots allocated to it than it requires to meet its completion time goal; (2) the function grows linearly between 0 and 0.5, when $s^j_{alloc} = s^j_{req}$ and thus in addition to the absolute minimum required for the next control cycle, the job is also allocated the estimated number of slots required over time to meet the completion time goal; and (3) the function grows linearly between 0.5 and 1.0, when $s^j_{alloc} = s^j_{pend}$ and thus all pending map tasks for this job are allocated a slot in the current control cycle.

Notice that u^M_j is a monotonically increasing utility function, with values in the range $(-\infty, 1]$. The intuition behind this function is that a job is unsatisfied ($u^M_j < 0$) when the number of slots allocated to map tasks is less than the minimum number required to meet the completion time goal of the job. Furthermore, the logarithmic shape of the function stresses the fact that it is critical for a job to make progress and therefore at least one slot must be allocated. A job is no longer unsatisfied ($u^M_j = 0$) when the allocation equals the requirement ($s^j_{alloc} = s^j_{req}$), and its satisfaction is positive ($u^M_j > 0$) and grows linearly when it gets more slots allocated than required. The maximum satisfaction occurs when all the pending tasks are allocated within the current control cycle ($s^j_{alloc} = s^j_{pend}$). The intuition behind u^R_j is that reduce tasks should start at the earliest possible time, so the shuffle sub-phase of the job (reducers pulling data produced by map tasks) can be fully pipelined with execution of map tasks. The logarithmic shape of this function indicates that any placement that does not run all reducers for a running job is unsatisfactory. The range of this function is $[-1, 0]$ and, therefore, it is used to subtract satisfaction of a job that, independently of the placement of map tasks, has unsatisfied demand for reduce tasks. If all the reduce tasks for a job are allocated, this function gets value 0 and thus, $u_j(P) = u^M_j(P^M)$.

Figure 5.17 shows the generic shape of the utility function for a job that requires at least 10 map tasks allocated during the next cycle ($s^j_{fit} = 10$), 15 map tasks concurrently over time ($s^j_{req} = 15$) to meet its completion time goal, has 35 map tasks ($s^j_{pend} = 35$) pending to be executed, and has been configured to run 10 reduce tasks ($r^j_{pend} = 10$), none of which have been started yet. On the X axis, a variable number of allocated slots for reduce tasks (r^j_{alloc}) is shown. On the Y axis, a variable number of allocated slots for map tasks (s^j_{alloc}) is shown. Finally, the Z axis shows the resulting utility value.

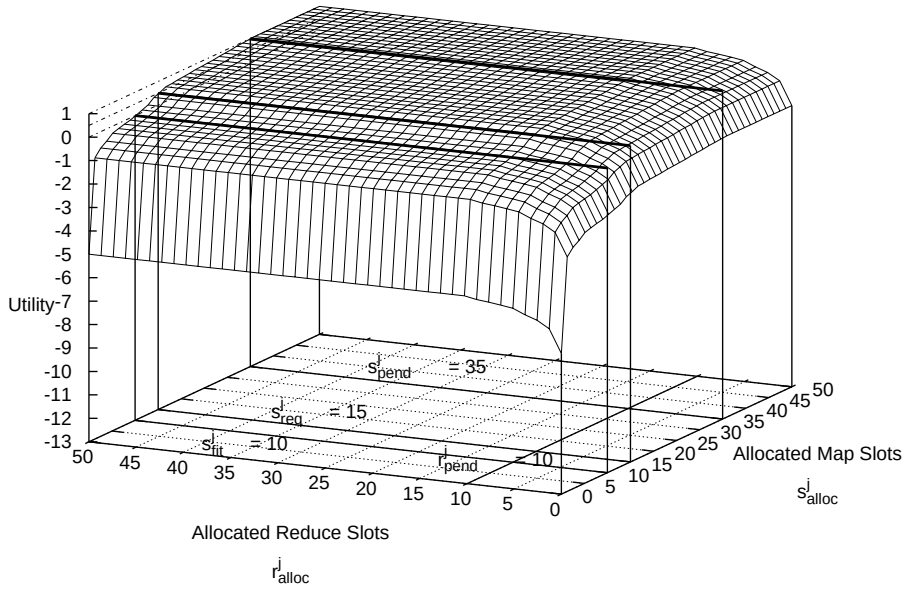


Figure 5.17: Shape of the Utility Function when $s^j_{fit} = 10$, $s^j_{req} = 15$ $s^j_{pend} = 35$, and $r^j_{pend} = 10$

5.3.4.4 *Job Placement Algorithm*

Given an application placement matrix P, a utility value can be calculated for each job in the system. The performance of the system can then be measured as an ordered vector of job utility values, U. The objective of the scheduler is to find a new placement P of jobs on tasktrackers that maximizes the global objective of the system, $U(P)$, which is expressed as follows:

$$\max \quad \min_j u_j(P) \tag{16}$$

$$\min \quad \Omega_{tt,r} - \sum_{tt}(\sum_j P_{j,tt}) * \Gamma_{j,r} \quad \forall_r \tag{17}$$

such that

$$\forall_{tt}\forall_r \quad (\sum_j P_{j,tt}) * \Gamma_{j,r} \leq \Omega_{tt,r} \tag{18}$$

$$and \quad \Omega_{n,r} = \Omega_{tt,r} + \Omega_{ds,r} \tag{19}$$

This optimization problem is a variant of the Class Constrained Multiple-Knapsack Problem. Since this problem is NP-hard, the scheduler adopts a heuristic inspired by [75]. While not described here, the Placement Algorithm itself is the same as that described in the previous chapter in Section 4.2.4, but using the proposed utility function described in Section 5.3.4.3.

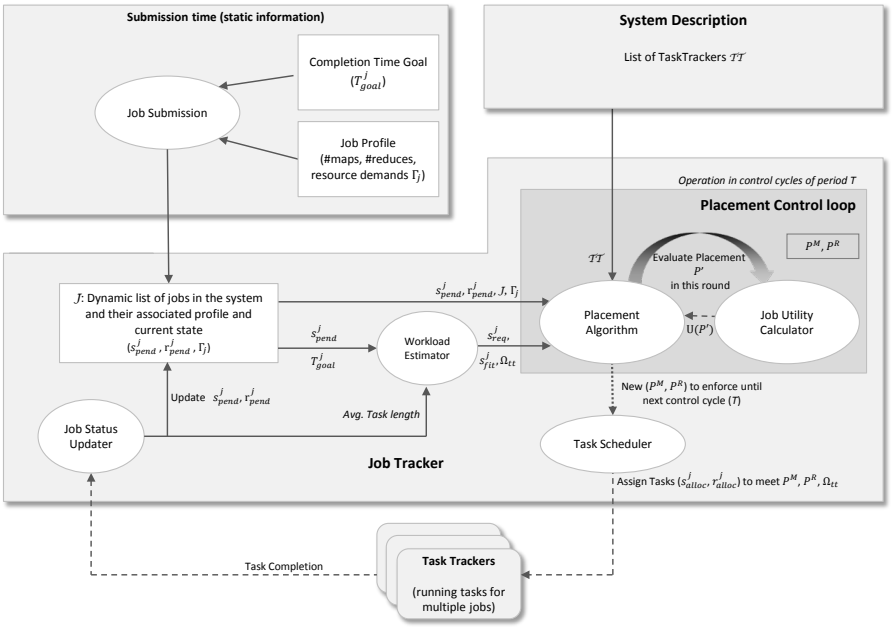


Figure 5.18: System architecture of the Reverse-Adaptive Scheduler

5.3.4.5 *Scheduler Architecture*

Figure 5.18 illustrates the architecture and operation the scheduler. The system consists of five components: Placement Algorithm, Job Utility Calculator, Task Scheduler, Job Status Updater and Workload Estimator. Each submission includes both the job's completion time goal (if one is provided) and its resource consumption profile.

Most of the logic behind the scheduler resides the utility-driven Placement Control Loop and the Task Scheduler. The former is responsible for producing placement decisions, while the latter is responsible for enforcing the decisions made by the former. The Placement Control Loop operates in control cycles of period T. Its output is a new placement matrix P that will be active until the next control cycle is reached (current time + T). The Task Scheduler is responsible for enforcing the placement decisions. The Job Utility Calculator calculates a utility value for an input placement matrix which is then used by the Placement Algorithm to choose the best placement choice available. Upon completion of a task, the TaskTracker notifies the Job Status Updater, which for any job j in the system, triggers an update of s^j_{pend} and r^j_{pend} in the job descriptor. The Job Status Updater also keeps track of the average task length observed for every job in the system, which is later used to estimate the completion time for each job. The Workload Estimator estimates the number of *map* tasks that should be allocated concurrently (s^j_{req}) to meet the completion time goal of each job, as well as the parameter S^j_{fit}.

In this work we concentrate on the estimation of the parameter S^j_{fit} that feeds the Placement Algorithm, as well as the performance

model used by the Job Utility Calculator. The major change in this architecture compared to the scheduler presented in the previous chapter in Section 4.2.4 is the introduction of the Workload Estimator, that not only estimates the demand for MapReduce tasks as did in previous work, but also provides estimates for the data-store resource consumption, derived from the calculation of $f(t)$.

5.3.5 *Evaluation*

This section includes the description of the experimental environment, including the simulation platform we have built, and the results from the experiments that explore the improvements of our scheduler compared to previous existing schedulers: the default FIFO scheduler, the Adaptive Scheduler described in [63], and the Reverse-Adaptive scheduler proposed in this section.

In Experiment 1 (Section 5.3.5.2) we consider the standard scenario in which MapReduce is the only workload running in the system and thus the performance of the scheduler should be similar to previous approaches. In Experiment 2 (Section 5.3.5.3) we introduce an additional workload in order to gain insight on how does the proposed scheduler perform in this kind of shared environment. And finally, Experiment 3 (Section 5.3.5.4) shows the impact that the burstiness of transactional workloads may have on the scheduler.

5.3.5.1 *Simulation Platform*

In order to simulate a shared environment, we built a system with two components. First, a workload generator to model the behaviour of multiple clients submitting jobs to the MapReduce cluster. And second, a server simulator to handle the workloads' submissions and schedule jobs depending on different policies.

The workload generator that describes the behaviour of clients takes the cluster configuration information as well as the desired workload parameters, and instantiates a number of jobs to meet those requirements. Table 5.6 describes the main workload configuration parameters used for the experiments. The dynamic availability of resources of the transactional workload ($f(t)$) is based on a real trace obtained from Twitter's frontend during an entire day, and has the same shape as that shown in Figures 5.14 and 5.15, with peak transactional utilization around hour 18. The distribution of MapReduce job lengths, which determines the number of tasks of each job, follows a lognormal that resembles the job sizes observed in known traces from Yahoo! and Facebook [20], but scaled down to a smaller number of jobs to fit into the 100-node cluster used during the simulations. For the distribution of deadlines factors we use a 3 different categories: tight (between 1.5x and 4x), regular (from 1.5x to 8x), and relaxed (from 1.5x to 12x).

PARAMETER	VALUE	DESCRIPTION
Cluster size	100	Total number of nodes in the system.
Node availability	$f(t)$	Function that represents the available number of nodes over time.
System load	$0.2 - 1.0$	Utilization of the Map-Reduce workload during the simulation. Determines the number of jobs.
Arrival distribution	Poisson: $n \approx 200 - 2500, \lambda \approx 1.5 - 15$	How arrivals are distributed over time. Depends on system load.
Job length distribution	Lognorm: $\mu = 62.0$, $\sigma = 15.5$	Determines the number of tasks of each job.
Deadline distribution	Uniform: $1.5x - [4, 8, 12]x$	Factor relative to completion time of jobs when executed in isolation.

Table 5.6: Main workload simulator configuration parameters.

In the experiments we simulate a total of 7 days, and in order to make sure the simulation is in a steady state we study and generate all the statistics for the 5 days in the middle, considering only jobs that either start or finish within that time window. For each experiment we obtain the averages and standard deviations of running 10 different simulated workloads generated with the same configuration parameters.

The simulation platform is written in Python using the NumPy and SciPy packages, and the Reverse-Adaptive implementation in particular is based on *splines* for fast, approximate curve fitting, interpolation and integration. While our proposal has not been optimized and is slower than the other schedulers we are simulating, it does not represent a performance issue for the amount of concurrent jobs that are usually executed in this kind of environment, specially considering MapReduce clusters run the scheduler on a dedicated machine and decisions are only made once per placement cycle, which is in the order of tens of seconds. In our experiments with hundreds to few thousands of jobs the scheduler is able to generate placement matrices in a time that always remains in the order of milliseconds. Our current implementation easily scales up to thousands of jobs and nodes.

Validation of the Simulation Platform

The MapReduce simulator allows recreating the conditions of complex environments in which multiple workloads with different characteristics are executed in the same set of machines as MapReduce.

In order to validate the accuracy of the MapReduce scheduling component of the simulator, we compare the simulator to a execution in a real environment. To that end we reproduce an experiment executed in an actual Hadoop MapReduce cluster, and convert the executed applications into a simulated workload.

The environment we simulate resembles the cluster of 61 nodes described in Section 3.5.2, and the set of applications is the same as that described in Section 3.5.1. In particular, we chose to reproduce one of the experiments described in Section 3.5.3. The synthetic mix of applications comprised in this workload represent a realistic scenario due to the different characteristics of each job. There are 4 different MapReduce applications that share the available resources during the execution. We configure each application with a particular completion time goal, derived from the completion time that each one of the applications achieves when executed in isolation. The completion time goals are set to be tight in order to show how the scheduler behaves when it's not possible to meet all completion time goals.

In the original experiment, the set of applications is as follows: a Simulator (J1) with a completion time goal of 6,000s (1.69X over its completion time in isolation); Wordcount (J2) is configured with a completion goal of 1,500s (1.18X); Sort (J3) is configured to complete after 650s (1.08X); and the Join executions (J4 and J5) are both configured with a completion time goal of 120s (1.18X).

The MapReduce simulator is limited to running a single phase, and so it's not possible to simulate both map and reduce phases at the same time. For this particular workload, the reduce phase of some of the applications (Simulator, Wordcount, and Join) is negligible due to either the absence of any real computation or the use of combiners. However, the Sort application involves significant computation during the reduce phase, and its deadline must be adapted since the simulator will not be able to simulate the reduce phase. Thus, while the deadline factor of the Simulator, Wordcount and Join applications remains the same for the simulation, the simulated Sort execution is slightly different since it only accounts for the map phase, which represents a fraction of the total completion time.

Figure 5.19 shows the evolution of how the simulator allocates tasks over time for each application. For the sake of clarity, jobs are grouped by application into different rows. Jobs J1 to J5 are submitted at times S1 to S5, and the completion time goals are D1 to D5. This execution using the scheduler ought to be compared to Figure 3.5 (page 48), which shows the evolution of the real execution. As it can be observed, the behaviour of applications during the simulation is similar to the

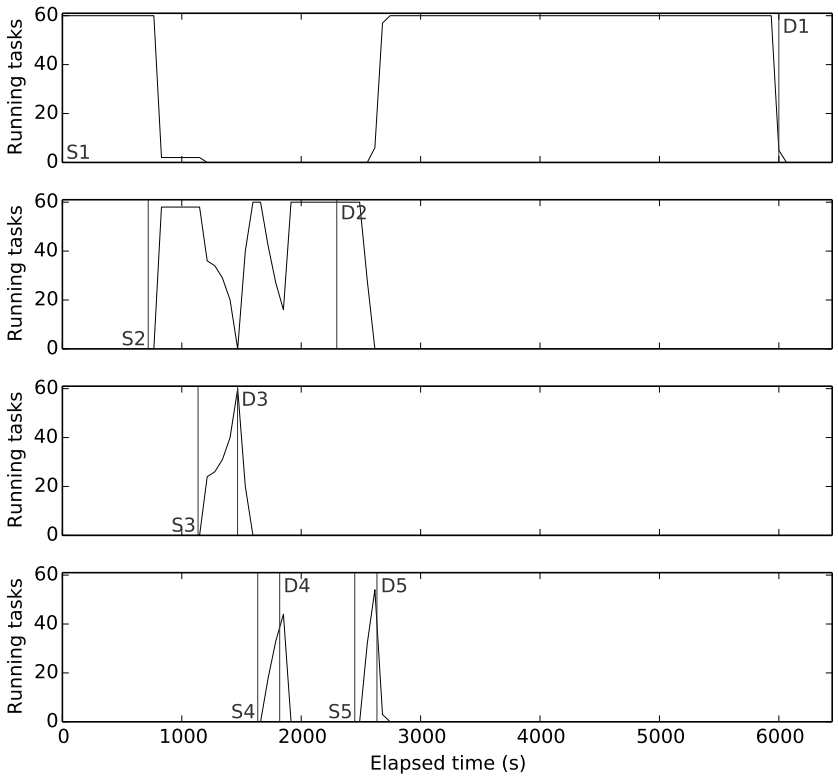


Figure 5.19: MapReduce simulation of the Adaptive Scheduler running the workload described in Section 3.5.3 and shown in Figure 3.5

real execution, and at a glance the only significant difference is the lack of reducers on J3.

Tables 5.7 and 5.8 provide more details about how far from the deadline is each one of the applications of the workload under both, real and simulated executions, as well as its execution time. The main differences can be found in J1 and J3 (Simulator application and Sort application, respectively). J1 represents an applications with irregular task lengths, a feature that the simulation is not able to capture correctly since it assumes regular task lengths. However, as shown in Table 5.7, the time beyond the deadline relative to the whole execution (which is longer than 6000s for J1) remains small. As for J3, the

	J1	J2	J3	J4	J5
Real execution	2%	17%	36%	38%	36%
Simulated execution	1%	18%	33%	42%	39%
Difference	-1%	+1%	-3%	+4%	+3%

Table 5.7: Relative time beyond deadline of each application of the workload under real and simulated environments

	J1	J2	J3	J4	J5
Real execution	6130.75	1816.08	427.05†	194.53	185.21
Simulated execution	6058.15	1821.94	395.73	207.93	197.23

Table 5.8: Execution time of each application of the workload under real and simulated environments, in seconds

† The execution time of the map phase is shown here to ease the comparison with the simulation; the full execution takes 1031.30 seconds.

difference can be explained due to the lack of support to simulate the reduce phase: the time beyond the deadline in the real execution is much higher because its execution is also longer, but again, when comparing the adapted relative time of each map phase, they are closer to each other (0.36 vs 0.33). Note that both J1 and J3 represent certain kinds of applications that are not considered as part of the workloads simulated in this chapter, but the simulator is still able to provide accurate results.

5.3.5.2 *Experiment 1: No Transactional Workload*

The goal of this experiment is to evaluate the scenario in which there is no additional workload other than MapReduce itself, and to assess that the scheduler does not introduce any flaw even in the worst-case scenario in which there is no transactional workload. It also represents the standard scenario considered by most MapReduce schedulers, which are only concerned with assigning tasks to a fixed number of nodes in the cluster.

To this end we disable the transactional workload on the simulator and make all resources available to the MapReduce workload. We then run the same experiments using the default FIFO scheduler, the Adaptive scheduler, and our proposed scheduler, the Reverse-Adaptive scheduler.

Figure 5.20 shows the percentage of missed deadlines for each scheduler under different configurations. On the first row the distribution of deadline factors assigned to jobs (meaning the time each job is given to complete) is uniformly distributed and ranges from a minimum of 1.5x to a maximum of 4x. On the second and third rows, the maximum deadline factor is increased to 8x and 12x respectively. Each row shows different load factors as well, which represent how busy is the cluster: from 0.2 (very small load) to 1.0 (fully loaded). As it can be observed, there is a significant difference between the default FIFO scheduler, which always misses more deadlines, and the other deadline-aware schedulers. Also, as expected, increasing the maximum deadline factor also has an impact on the number of missed deadlines on all schedulers, but even more so on the Adaptive and

Reverse-Adaptive schedulers since that gives them more flexibility and a higher chance of distributing the execution of jobs.

On the other hand, in this scenario the Reverse-Adaptive scheduler performs exactly like the Adaptive scheduler under all configurations since it is not able to leverage the information about the characteristics of other non-MapReduce workloads in order to improve its performance. But it also shows that under no circumstances will the Reverse-Adaptive scheduler perform worse than previous deadline-aware schedulers in terms of missed deadlines.

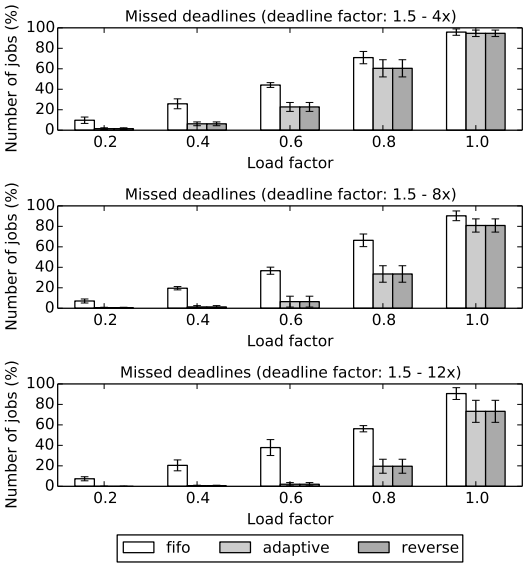


Figure 5.20: Experiment 1: No transactional workload.

5.3.5.3 *Experiment 2: Transactional Workload*

In this experiment we evaluate the Reverse-Adaptive scheduler in the presence of transactional workloads, and compare it to other schedulers, showing also additional metrics that help understand the behaviour of our algorithm. In particular, we study the same workload under different load levels: from 0.2 (low load) to 0.8 (high load); and also with different deadline factor distributions, ranging from 1.5x–4x to 1.5–12x. The transactional workload changes the availability of resources over time, and is based on a real trace as described in 5.3.5.1.

Figure 5.21 shows the results for each deadline factor distribution: 1.5x – 4x (Figure 5.21a), 1.5x – 8x (Figure 5.21b), 1.5x – 12x (Figure 5.21c). And each figure shows the number of jobs that miss their deadline (1st row), time beyond deadline for jobs that miss their deadline (2nd row), and distance to deadline for jobs that meet their deadline (3rd row). For this experiment we also run a fourth execution of the simulator with a different optimization goal that only takes into account minimizing the number of missed deadlines, and does not

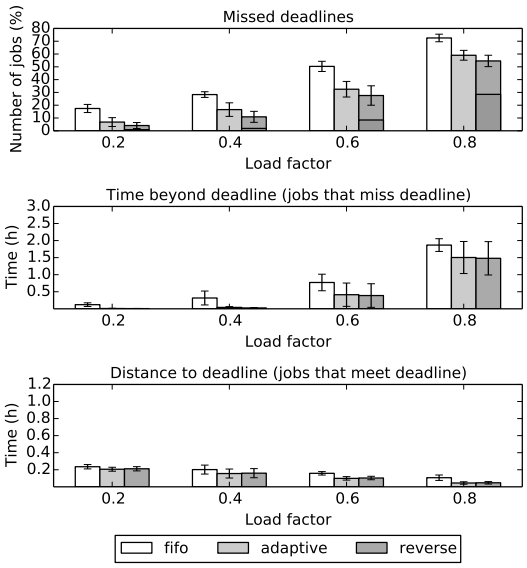
consider any fairness goals found in other schedulers. It is shown in the first row of Figures 5.21a to 5.21c as a horizontal line on the Reverse-Adaptive scheduler bars. We use these as a reference to distinguish why are schedulers missing deadlines, as it marks the minimum amount of jobs that will miss their deadline independently of the policies of the scheduler.

As it can be observed in the three figures, introducing a dynamic transactional workload allows the scheduler to improve the number of missed deadlines without a significant impact on other metrics. As shown in Figure 5.21a, which represents executions when running with a tight deadline factor distribution between 1.5x and 4x, the number of deadlines missed by the Reverse-Adaptive scheduler is always noticeable lower than that of the Adaptive and FIFO schedulers (1st row), while the time beyond deadline is only slightly lower (2nd row), and distance to deadline remains mostly the same with very small variations (3rd row). These results remain the same with more relaxed deadline factors as shown in Figure 5.21b and 5.21c. Notice that the improvement in terms of percentage of missed deadlines with the Reverse-Adaptive scheduler compared to other schedulers is similar despite the different deadline factors. This is basically because in these three scenarios the actual chance of improving is similar, as shown by the horizontal lines marking the percentage of jobs that will be missed for certain.

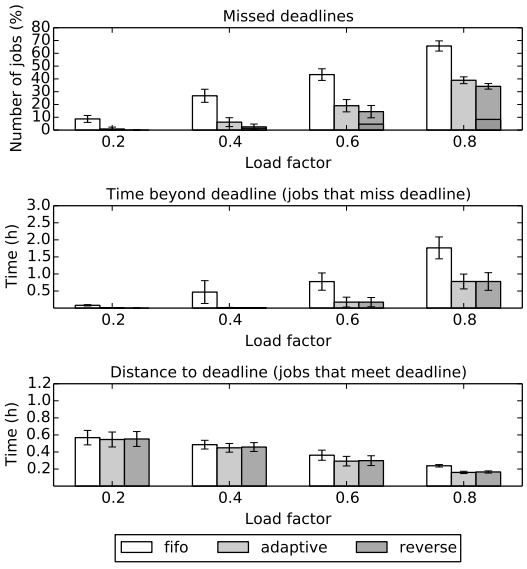
5.3.5.4 *Experiment 3: Burstiness of Transactional Workload*

This experiment explores the impact of transactional workload burstiness on the scheduler. While the previous experiment shows that the scheduler is able to leverage the characteristics of the transactional workload to improve the performance of the scheduler, in this experiment we show how the shape of the availability function affects the chances of improving the overall results. In particular, burstiness in this scenario means variability between the highest and lowest points of the availability function. Figure 5.22 shows the different burstiness *levels* evaluated in this experiment: from high burstiness (level 3) to low burstiness (level 1). As a reference to compare to previous executions, note that Experiment 1 has no burstiness at all, while Experiment 2 represents high burstiness (equivalent to level 3).

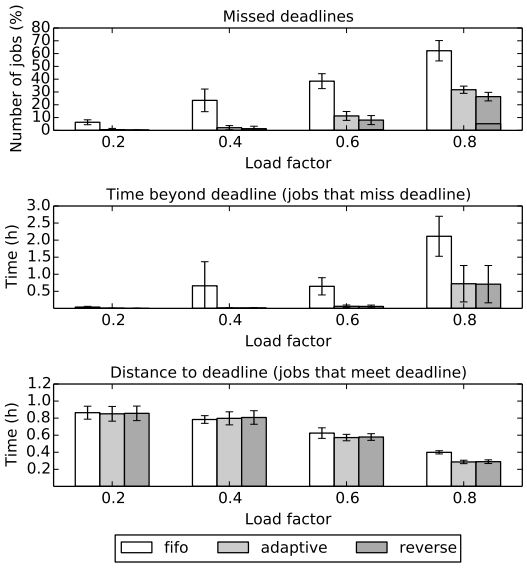
Figure 5.23 shows the number of jobs that miss their deadline when running with the different burstiness levels. To simplify this experiment we only execute the simulator with a medium deadline factor (1.5x to 8x). As it can be observed, the more bursty the availability function, the more likely is the scheduler able to improve its performance and lower the amount of missed deadlines. This is basically because the higher variability of high burstiness levels makes the available resources less predictable and estimations less accurate.



(a)



(b)



(c)

Figure 5.21: Experiment 2: Scheduling with transactional workload. Deadline factors: 1.5x – 4x (a), 1.5x – 8x (b), 1.5x – 12x (c).

However, with lower burstiness (and thus variability), availability is not as likely to change, so estimations are still relatively accurate.

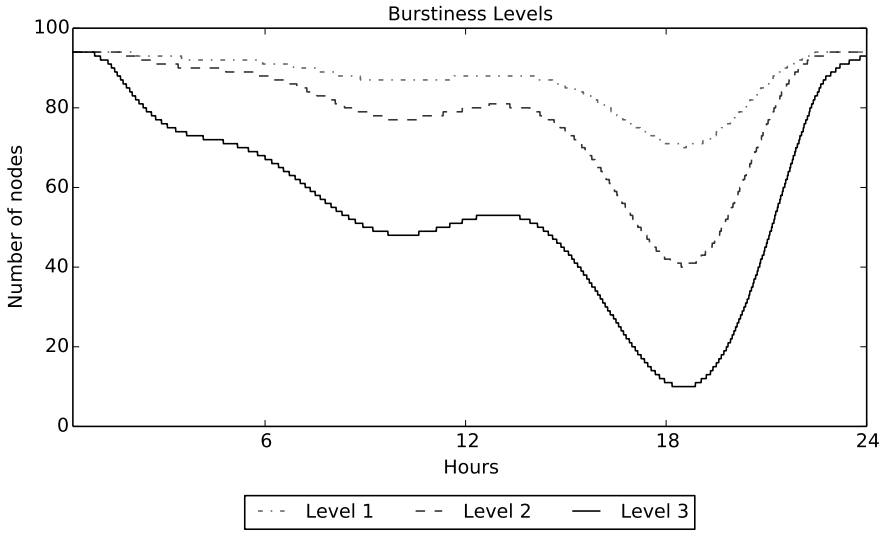
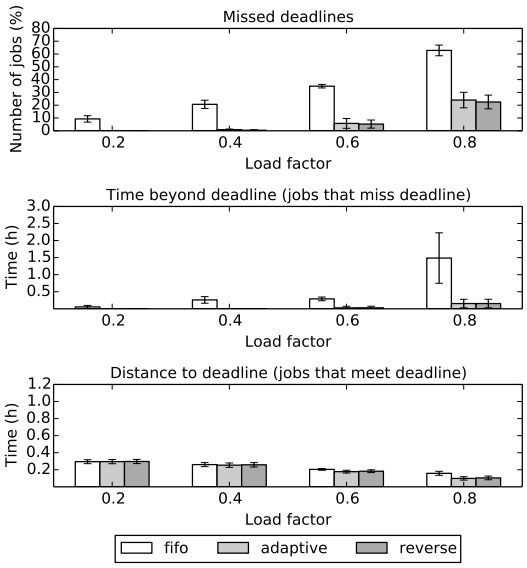


Figure 5.22: Experiment 3: Burstiness level classification.

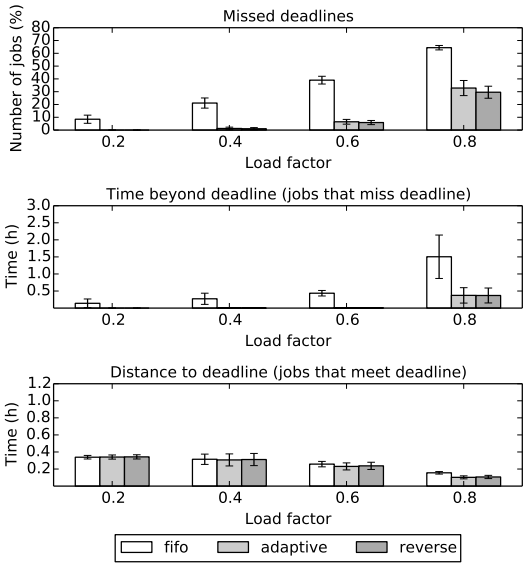
5.3.6 *Related Work*

Much work has been done in the space of scheduling for MapReduce. Since the number of resources and slots in a Hadoop cluster is fixed through out the lifetime of the cluster, most of the proposed solutions can be reduced to a variant of the *task-assignment* or *slot-assignment* problem. The Capacity Scheduler [88] is a pluggable scheduler developed by Yahoo! which partitions resources into pools and provides priorities for each pool. Hadoop's Fair Scheduler [89] allocates equal shares to each tenant in the cluster. All these schedulers are built on top of the same resource model and do not support high-level user-defined goals nor dynamic availability in shared environements.

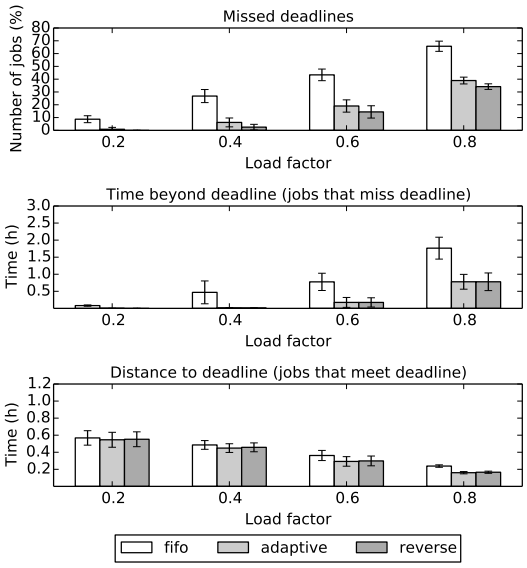
The performance of MapReduce jobs has attracted much interest in the Hadoop community. Recently, there has been increasing interest in user-centric data analytics, as proposed in [59], which introduces a scheduling scheme that enables soft-deadline support for MapReduce jobs. It differs from the presented proposal in that it does not take into consideration neither the resources of the system nor other workloads. Similarly, Flex [86] is a scheduler proposed as an add-on to the Fair Scheduler to provide Service-Level-Agreement (SLA) guarantees. More recently, Aria [82] introduces a novel resource management framework that consists of a job profiler, a model for MapReduce jobs and a SLO-scheduler based on the Earliest Deadline First scheduling strategy. Flex and Aria are both slot-based and therefore suffers from the same limitations we mentioned earlier.



(a)



(b)



(c)

Figure 5.23: Experiment 3: Execution with different burstiness: level 1 (a), level 2 (b), and level 3 (c); deadline factor from 1.5x to 8x.

New platforms have been proposed to mix MapReduce frameworks like Hadoop with other kinds of workloads. Mesos [39] intends to improve cluster utilization on shared environments, but is focused on batch-like and HPC instead of transactional workloads. Finally, the Hadoop community has also recognized the importance of developing a resource-aware scheduling for MapReduce. [12] outlines the vision behind the Hadoop scheduler of the future. The framework proposed introduces a resource model consisting of a 'resource container' which is fungible across jobs. We think that our proposed resource management techniques can be leveraged within this framework to enable better resource management.

5.4 SUMMARY

The first part of this Chapter presented a technique to provide stronger isolation support on top of distributed key-value stores, and implemented it for Apache Cassandra.

The proposed approach takes advantage of the fact that one of the major structures used to persist data in this kind of stores are SSTables, which are immutable. This proposal modifies Cassandra so as to keep SSTables when requested by concurrently running transactions, effectively allowing multi-versioned concurrency control for read operations on Cassandra in the form of snapshots. As shown in our evaluation, the new version of Cassandra with Snapshotted Read support is able to read from snapshots with a low impact on read latency and the overall performance of the system. While regular reads are slightly slower on the new version of Cassandra, operations that read from a snapshot are sometimes faster due to its limited scope.

This approach to improve the isolation capabilities of distributed key-value stores without compromising its performance is specially interesting in the environments in which these stores are nowadays executed, which tend to involve a range of technologies on the back-end side instead of a single database solution, and different applications and workloads running at the same time sharing and processing the same data.

The second part of this Chapter presented the Reverse-Adaptive Scheduler, which introduces a novel resource management and job scheduling scheme for MapReduce when executed in shared environments along with other kinds of workloads. The proposed scheduler is capable of improving resource utilization and job performance. The model introduced here allows for the formulation of a placement problem which is solved by means of a utility-driven algorithm. This algorithm in turn provides the scheduler with the adaptability needed to respond to changing conditions in resource demand and availability of resources.

The scheduler works by estimating the need of resources that should be allocated to each job, but in a more proactive way than previously existing work, since the estimation takes into account the expected availability of resources. In particular, the proposed algorithm consists of two major steps: reversing the execution of the workload and generating the current placement of tasks. Reversing the execution of the workload involves creating an estimated placement of the full workload over time, assigning tasks in the opposite direction: starting at the desired end state and finishing at the current state. The *reversed* placement is used as an estimation to know how many tasks are left at the current state, which allows the scheduler to determine what's the need of tasks for each job and how should they share the available resources. The presented scheduler relies on existing profiling information based on previous executions of jobs to make scheduling and placement decisions.

The work described in this chapter has resulted in the following main publications:

[65] Jordà Polo, David Carrera, Yolanda Becerra, Malgorzata Steinder, Mike Spreitzer, Jordi Torres, and Eduard Ayguadé. Enabling Distributed Key-Value Stores with Low Latency-Impact Snapshot Support. In *Proceedings of the 12th IEEE International Symposium on Network Computing and Applications (NCA 2013)*, Boston, MA, USA, 2013. IEEE Computer Society

[66] Jordà Polo, David Carrera, Yolanda Becerra, Jordi Torres, Eduard Ayguadé, and Malgorzata Steinder. Adaptive MapReduce Scheduling in Shared Environments. In *Proceedings of the 14th IEEE ACM International Symposium On Cluster, Cloud And Grid Computing (CCGrid 2014)*, Chicago, IL, USA, 2014. IEEE Computer Society

6

CONCLUSIONS AND FUTURE WORK

6.1 CONCLUSIONS

6.1.1 *Scheduling with Time Constraints*

The first contribution of this thesis, presented in Chapter 3, consists of a performance model for multi-job MapReduce workloads and a scheduling algorithm that leverages the model and allows management with time constraints. The effectiveness of the scheduler is demonstrated through a prototype implementation and evaluation on the Hadoop platform.

MapReduce jobs are composed of a large number of tasks known in advance during the job initialization phase (when the input data is split into smaller chunks). This characteristic can be leveraged to dynamically estimate the progress of MapReduce applications at run time. Therefore, adaptive schedulers can be developed to allow users to provide high-level performance objectives, such as completion time goals.

The proposed scheduler dynamically adjusts the allocation of available execution slots across jobs so as to meet their completion time goals, provided at submission time. The system continuously monitors the average task length for all jobs in all nodes, and uses this information to calculate and adjust the expected completion time for all jobs. Beyond completion time objectives, the presented scheduler also observes two additional high-level performance goals: first, it enforces data locality when possible, reducing the total volume of network traffic for a given workload; and secondly, it is also able to deal with hybrid machines composed of generic processors and hardware accelerators that can carry specialized tasks.

Data locality is increased by introducing simple yet effective mechanisms into the task scheduler, while still meeting high-level performance goals. Improved locality does not necessarily result in better performance for the individual jobs, but it is always correlated with lower network consumption.

Heterogeneous nodes and processors offer certain advantages to some MapReduce workloads, providing specialized cores that can perform critical tasks more efficiently. Exploiting such hardware infrastructure requires some kind of awareness on the part of the task scheduler, providing hardware affinity when necessary. Real-time monitoring of tasks allows the scheduler to evaluate the real benefits of running each workload on different platforms, and the scheduler is able decide the best distribution of tasks accordingly. Depending on the individual performance goals of each job, and on the availability of generic and hardware-specific code for each application, the scheduler is able to decide which version to run on top of the available hardware. Low-level programming languages that support different parallel platforms can provide an even greater advantage in these heterogeneous scenarios.

To evaluate the proposed scheduler, the prototype is deployed in two clusters of machines running Hadoop: one of them equipped with general purpose processors, and another one enabled with hardware accelerators. A number of experiments are executed to demonstrate the effectiveness of the proposed technique with regards to the three objectives: completion time goals, data locality and support for hardware accelerators. The results are also compared with another state-of-the-art Hadoop scheduler, and show how the proposed scheduler enables users to define and predict the performance of the system more accurately. Compared to other schedulers, the prototype is able to improve the performance of MapReduce workloads when these are composed of jobs with different priorities, jobs with a high network consumption, and also jobs that benefit from specialized hardware.

6.1.2 *Scheduling with Space and Time Constraints*

The second contribution of this thesis, described in Chapter 4, is a new resource model for MapReduce and a scheduler based on a resource-aware placement algorithm that leverages the proposed model. The scheduler, built upon the one presented in the first contribution, is aware of the underlying hardware as well as the characteristics of the applications, and is capable of improving cluster utilization while still being guided by job performance metrics.

The foundation of this scheduler is a resource model based on a new abstraction, namely the "job slot". This model allows for the formulation of a placement problem which the scheduler solves by

means of a utility-driven algorithm. This algorithm in turn provides the scheduler with the adaptability needed to respond to changing conditions in resource demand.

The presented scheduler relies on existing profiling information based on previous executions of jobs to make scheduling and placement decisions. Profiling of MapReduce jobs that run periodically on data with similar characteristics is an easy task, which has been used by many others in the community in the past. However, this proposal pioneers a novel technique for scheduling reduce tasks by incorporating them into the utility function driving the scheduling algorithm. The proposed approach works well in most scenarios, but it may need to rely on preempting reduce tasks to release resources for jobs with higher priority. The scheduler considers three resource capacities: CPU, memory and I/O, but it could be easily extended to incorporate additional resources of the tasktrackers.

A prototype of the scheduler has been implemented on top of Hadoop, and its source code is publicly available at [57]. In order to evaluate the prototype, a number of experiments have been executed in a real cluster driven by representative MapReduce workloads, and compared to a state-of-the-art scheduler. The experiments show the effectiveness of this proposal, which is able to improve cluster utilization in multi-job MapReduce environments, even in the presence of completion time constraints. The results also show the benefits of using simple job profiles, which enable the scheduler with the required knowledge to improve its performance. To the best of our knowledge the proposed scheduler is the first scheduling framework to use a new resource model in MapReduce and also the first to leverage resource information to improve the utilization of resources in the system while still meeting completion time goals on behalf of users.

6.1.3 *Scheduling with Space and Time Constraints in Shared Environments*

Finally, the third contribution of this thesis, presented in Chapter 5, addresses two related problems found in shared environment scenarios with MapReduce. First, a scheduler and performance model for shared environments that allows an integrated management of resources in the presence of other non-MapReduce workloads, and second, the necessary mechanism to allow the shared data store to be used for both, transactional and analytics workloads.

The proposed scheduler is able to improve resource utilization across machines while observing completion time goals, taking into account the resource demands of non-MapReduce workloads, and assuming that the amount of resources made available to the MapReduce applications is dynamic and variable over time. This is achieved thanks to a new algorithm that provides a more proactive

approach for the scheduler to estimate the need of resources that should be allocated to each job.

A prototype of the scheduler has been evaluated in a simulated environment and compared to other MapReduce schedulers. Experiments driven by representative MapReduce workloads demonstrate the effectiveness of this proposal, which is capable of improving resource utilization and job performance in the presence of other non-MapReduce workloads. To the best of our knowledge this is the first scheduling framework to take into account other workloads, such as transactional workloads, in addition to leveraging resource information to improve the utilization of resources in the system and meet completion time goals on behalf of users.

On the other hand, the proposal to enable the storage with additional isolation capabilities is implemented on top of a distributed key-value store, which is used as a good middle ground between traditional databases and distributed filesystems. This approach takes advantage of the fact that one of the major structures used to persist data in this kind of stores is immutable, which helps minimize the performance impact of supporting the additional operations required to run different kinds of workloads.

A prototype of this proposal has been implemented and evaluated on top of the key-value store Cassandra. As shown by the experiments run on a real cluster using a well-known benchmark, the prototype is capable of providing snapshots without a significant penalty on its performance. This approach then effectively allows a data stores to be used for analytics and provide support for some transactional operations at the same time.

6.2 FUTURE WORK

The work performed in this thesis opens several interesting proposals that could be explored as part of future work.

- As part of this thesis, a simple profile is used to determine resource utilization based on previous executions (offline), and a model based on task allocation is used to estimate the performance of jobs (online). While this approach has proved to be good for most applications, there is still room for improvement with regards to the characterization of jobs in order to improve the management of MapReduce workloads. A more fine-grained profile could help characterize the details of each phase, and would make it easier to avoid certain bottlenecks. Another option to improve the performance management would be reusing data and statistics from previous executions. This is not trivial since the same job may be executed using a completely different input and thus potentially have different behaviours.

But ideally, repeated executions would provide more data over time.

- The scheduler presented in this thesis is aware of resources such as CPU, disk IO and memory consumption. It would be trivial to add additional resources and hardware constraints to the placement algorithm as long as they affect a single node, e.g. disk capacity. But there are certain resources that require a more thoughtful approach since they depend not only on the MapReduce job itself, but also on other variables that require cluster-level awareness. An example of one such resource is network utilization, which can change significantly depending on where is data located in the underlying distributed filesystem.

- Reduce tasks have traditionally been one of the most challenging aspects of MapReduce scheduling, yet it is still sometimes overlooked. Reduce tasks are hard to schedule because they are meant to be run as a single non-preemptive *wave* of tasks, and they actually consist of different sub-phases: 1) copying all the data from the map tasks, and 2) performing the actual reduce function. Most schedulers launch reduce tasks as soon as possible in order to start the data transfer beforehand. However, this approach is not perfect since reduce tasks will keep running until all maps are completed. It should be possible to improve the scheduling of reduce tasks by either introducing some kind of separation between its sub-phases, or finding the optimal execution that minimizes its waiting time.

- The simplicity of MapReduce is one of the keys to its widespread adoption, but sometimes it also forces users to use certain workarounds to overcome some of its limitations. When the two-phase map and reduce model is not enough to produce the desired results, some applications resort to *chaining* multiple MapReduce jobs as a sequence of operations. These are becoming even more common with high-level abstraction layers on top of Hadoop (e.g. Pig [50] or Hive [78]), which tend to transform programs into sequences of back-to-back MapReduce jobs. This kind of applications pose significant issues for MapReduce schedulers since they should be thought of as a single unit, but in practice become a number, sometimes unknown, of jobs. A more thoughtful study and characterization of the dataflow of sequential applications would be useful for any scheduler.

BIBLIOGRAPHY

[1] Adaptive Scheduler. https://issues.apache.org/jira/browse/MAPREDUCE-1380.

[2] *NASA Nebula project*. URL http://nebula.nasa.gov.

[3] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proc. VLDB Endow.*, 2(1):922–933, August 2009. ISSN 2150-8097.

[4] Rajendra Akerkar, editor. *Big Data Computing*. Taylor & Francis Group – CRC Press, 2013. ISBN 978-1-46-657837-1. URL http://www.taylorandfrancis.com/books/details/9781466578371/.

[5] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in Map-Reduce clusters using Mantri. In *9th USENIX Conference on Operating Systems Design and Implementation*, pages 1–16, Berkeley, USA, 2010.

[6] Apache Software Foundation. HDFS Architecture, 2009. URL http://hadoop.apache.org/common/docs/current/hdfs_design.html.

[7] Apache Software Foundation. Hadoop on Demand, 2009. URL http://hadoop.apache.org/core/docs/r0.20.0/hod_user_guide.html.

[8] Apache Software Foundation. Hadoop MapReduce, 2009. URL http://hadoop.apache.org/.

[9] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: a Berkeley view of cloud computing. Technical report, University of California at Berkeley, February 2009. URL http://berkeleyclouds.blogspot.com/2009/02/above-clouds-released.html.

[10] Michael Armbrust, Kristal Curtis, Tim Kraska, Armando Fox, Michael J. Franklin, and David A. Patterson. Piql: success-tolerant query processing in the cloud. *Proc. VLDB Endow.*, 5 (3):181–192, November 2011. ISSN 2150-8097.

[11] Arun C. Murthy, Chris Douglas, Mahadev Konar, Owen O'Malley, Sanjay Radia, Sharad Agarwal, Vinod K V. Architecture of Next Generation Apache Hadoop MapReduce Framework. URL https://issues.apache.org/jira/secure/attachment/12486023/.

[12] Arun Murthy. Next Generation Hadoop Scheduler. URL http://developer.yahoo.com/blogs/hadoop/posts/2011/03/mapreduce-nextgen-scheduler/.

[13] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research*, 2011.

[14] Luiz André Barroso, Jimmy Clidaras, and Urs Holzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines, second edition. *Synthesis Lectures on Computer Architecture*, 8(3):1–154, 2013. doi: 10.2200/S00516ED2V01Y201306CAC024. URL http://www.morganclaypool.com/doi/abs/-10.2200/S00516ED2V01Y201306CAC024.

[15] Yolanda Becerra, Vicenç Beltran, David Carrera, Marc Gonzalez, Jordi Torres, and Eduard Ayguadé. Speeding up distributed mapreduce applications using hardware accelerators. In *ICPP '09: Proceedings of the 2009 International Conference on Parallel Processing*, pages 42–49, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3802-0.

[16] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, SIGMOD '95, pages 1–10, New York, NY, USA, 1995. ACM. ISBN 0-89791-731-6. doi: 10.1145/223784.223785.

[17] Jacek Błażewicz, Maciej Machowiak, Jan Węglarz, Mikhail Y Kovalyov, and Denis Trystram. Scheduling malleable tasks on parallel processors to minimize the makespan. *Annals of Operations Research*, 129(1-4):65–80, 2004. ISSN 0254-5330.

[18] David Carrera, Malgorzata Steinder, Ian Whalley, Jordi Torres, and Eduard Ayguadé. Enabling resource sharing between transactional and batch workloads using dynamic application placement. In *Middleware '08: Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, pages 203–222, New York, NY, USA, 2008. Springer-Verlag New York, Inc. ISBN 3-540-89855-7.

[19] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008. ISSN 0734-2071. doi: 10.1145/1365815.1365816. URL http://doi.acm.org/10.1145/1365815.1365816.

[20] Yanpei Chen, Archana Sulochana Ganapathi, Rean Griffith, and Randy H. Katz. A methodology for understanding mapreduce performance under diverse workloads. Technical Report UCB/EECS-2010-135, EECS Department, University of California, Berkeley, Nov 2010. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-135.html.

[21] Byung-Gon Chun, Gianluca Iannaccone, Giuseppe Iannaccone, Randy Katz, Gunho Lee, and Luca Niccolini. An energy case for hybrid datacenters. In *Workshop on Power Aware Computing and Systems (HotPower'09)*, Big Sky, MT, USA, 2009, 10/2009 2009.

[22] Jonathan Cohen. Graph twiddling in a mapreduce world. *Computing in Science and Engineering*, 11(4):29–41, 2009. ISSN 1521-9615. doi: 10.1109/MCSE.2009.120. URL http://dx.doi.org/10.1109/MCSE.2009.120.

[23] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0036-0. doi: 10.1145/1807128.1807152.

[24] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI '04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, December 2004. URL http://labs.google.com/papers/mapreduce.html.

[25] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5. doi: 10.1145/1294261.1294281. URL http://doi.acm.org/10.1145/1294261.1294281.

[26] Jaideep Dhok and Vasudeva Varma. Using pattern classification for task assignment. URL http://researchweb.iiit.ac.in/~jaideep/jd-thesis.pdf.

[27] Fangpeng Dong and Selim G Akl. Scheduling algorithms for grid computing: State of the art and open problems. *School of Computing, Queen's University, Kingston, Ontario*, 2006.

[28] James Corbett et all. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6.

[29] Dror Feitelson. Job scheduling in multiprogrammed parallel systems. *IBM Research Report*, 19790, 1997.

[30] Dror G. Feitelson and Larry Rudolph. Parallel job scheduling: Issues and approaches. In *JSSPP*, pages 1–18, 1995.

[31] Dror G Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C Sevcik, and Parkson Wong. Theory and practice in parallel job scheduling. In *Job scheduling strategies for parallel processing*, pages 1–34. Springer Berlin Heidelberg, 1997.

[32] Dror G Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Parallel job scheduling–a status report. In *Job Scheduling Strategies for Parallel Processing*, pages 1–16. Springer Berlin Heidelberg, 2005.

[33] Ian Foster and Ian Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 2002.

[34] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International journal of high performance computing applications*, 15(3):200–222, 2001.

[35] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/1165389.945450. URL http://labs.google.com/papers/gfs.html.

[36] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. Scalable consistency in scatter. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 15–28, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043559.

[37] M. Gschwind, P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. A novel SIMD architecture for the cell heterogeneous chip-multiprocessor. 2005.

[38] Herodotos Herodotou and Shivnath Babu. Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. *VLDB*, 2010.

[39] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, pages 22–22. USENIX Association, 2011.

[40] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-636-3.

[41] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-636-3.

[42] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 261–276, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. URL http://www.sigops.org/sosp/sosp09/papers/isard-sosp09.pdf.

[43] Predrag R. Jelenkovic, Xiaozhu Kang, and Jian Tan. Adaptive and scalable comparison scheduling. In *SIGMETRICS'07*, NY, USA. ACM. ISBN 978-1-59593-639-4.

[44] F. Junqueira, B. Reed, and M. Yabandeh. Lock-free transactional support for large-scale storage systems. In *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*, pages 176 –181, june 2011. doi: 10.1109/DSNW.2011.5958809.

[45] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2): 35–40, April 2010. ISSN 0163-5980. doi: 10.1145/1773912.1773922. URL http://doi.acm.org/10.1145/1773912.1773922.

[46] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A case for flash memory SSD in enterprise database applications. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 1075–1086, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi: http://doi.acm.org/10.1145/1376616.1376723.

[47] Joseph YT Leung. *Handbook of scheduling: algorithms, models, and performance analysis*. CRC Press, 2004.

[48] David A Lifka. The anl/ibm sp scheduling system. In *Job Scheduling Strategies for Parallel Processing*, pages 295–303. Springer, 1995.

[49] Los Alamos National Laboratory. *High-Performance Computing: Roadrunner*. URL http://www.lanl.gov/roadrunner/.

[50] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[51] Owen O'Malley. Terabyte sort on Apache Hadoop, 2008. URL http://www.hpl.hp.com/hosted/sortbenchmark/YahooHadoop.pdf.

[52] Owen O'Malley and Arun Murthy. Winning a 60 second dash with a yellow elephant, 2009. URL http://developer.yahoo.com/blogs/hadoop/Yahoo2009.pdf.

[53] Giovanni Pacifici, Wolfgang Segmuller, Mike Spreitzer, and Asser Tantawi. Dynamic estimation of cpu demand of web traffic. In *Proceedings of the 1st international conference on Performance evaluation methodolgies and tools*, valuetools '06, New York, NY, USA, 2006. ACM. ISBN 1-59593-504-5. doi: 10.1145/1190095.1190128. URL http://doi.acm.org/10.1145/1190095.1190128.

[54] Giovanni Pacifici, Wolfgang Segmuller, Mike Spreitzer, and Asser N. Tantawi. Dynamic estimation of cpu demand of web traffic. In *VALUETOOLS*, page 26, 2006.

[55] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, SIGMOD '09, pages 165–178, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-551-2. doi: 10.1145/1559845.1559865.

[56] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[57] Jordà Polo. Adaptive Scheduler, 2009. URL https://issues.apache.org/jira/browse/MAPREDUCE-1380.

[58] Jordà Polo. *Big Data Computing*, chapter Big Data Processing with MapReduce. In Akerkar [4], 2013. ISBN 978-1-46-657837-1. URL http://www.taylorandfrancis.com/books/details/9781466578371/.

[59] Jordà Polo, David Carrera, Yolanda Becerra, Malgorzata Steinder, and Ian Whalley. Performance-driven task co-scheduling for MapReduce environments. In *Network Operations and Management Symposium, NOMS*, pages 373–380, Osaka, Japan, 2010.

[60] Jordà Polo, David Carrera, Yolanda Becerra, Malgorzata Steinder, and Ian Whalley. Performance-driven task co-scheduling for MapReduce environments. In *Proceedings of the 12th IEEE/IFIP Network Operations and Management Symposium*, pages 373–380, Osaka, Japan, 2010.

[61] Jordà Polo, David Carrera, Yolanda Becerra, Jordi Torres, and Eduard Ayguadé. Performance Management of Accelerated MapReduce Workloads in Heterogeneous Clusters. In *ICPP '10: Proceedings of the 39th IEEE/IFIP International Conference on Parallel Processing*, San Diego, CA, USA, 2010.

[62] Jordà Polo, David Carrera, Yolanda Becerra, Jordi Torres, Eduard Ayguadé, Malgorzata Steinder, and Ian Whalley. Performance-driven task co-scheduling for MapReduce environments. In *Network Operations and Management Symposium, NOMS*, pages 373–380, Osaka, Japan, 2010.

[63] Jordà Polo, Claris Castillo, David Carrera, Yolanda Becerra, Ian Whalley, Malgorzata Steinder, Jordi Torres, and Eduard Ayguadé. Resource-Aware Adaptive Scheduling for MapReduce Clusters. In *ACM IFIP USENIX 12th International Middleware Conference*, pages 187–207, Lisbon, Portugal, 2011. Springer. ISBN 978-3-642-25820-6. doi: 10.1007/978-3-642-25821-3_10.

[64] Jordà Polo, Yolanda Becerra, David Carrera, Malgorzata Steinder, Ian Whalley, Jordi Torres, and Eduard Ayguadé. Deadline-Based MapReduce Workload Management. *IEEE Transactions on Network and Service Management*, pages 1–14, 2013-01-08 2013. ISSN 1932-4537.

[65] Jordà Polo, David Carrera, Yolanda Becerra, Malgorzata Steinder, Mike Spreitzer, Jordi Torres, and Eduard Ayguadé. Enabling Distributed Key-Value Stores with Low Latency-Impact Snapshot Support. In *Proceedings of the 12th IEEE International Symposium on Network Computing and Applications (NCA 2013)*, Boston, MA, USA, 2013. IEEE Computer Society.

[66] Jordà Polo, David Carrera, Yolanda Becerra, Jordi Torres, Eduard Ayguadé, and Malgorzata Steinder. Adaptive MapReduce Sched-

uling in Shared Environments. In *Proceedings of the 14th IEEE ACM International Symposium On Cluster, Cloud And Grid Computing (CCGrid 2014)*, Chicago, IL, USA, 2014. IEEE Computer Society.

[67] Sean Quinlan. GFS: Evolution on fast-forward. *ACM Queue*, 2009. URL http://portal.acm.org/ft_gateway.cfm?id=1594206&type=pdf.

[68] Dharavath Ramesh, Amit Kumar Jain, and Chiranjeev Kumar. Implementation of atomicity and snapshot isolation for multi-row transactions on column oriented distributed databases using rdbms. In *Communications, Devices and Intelligent Systems, 2012 International Conference on*, pages 298 –301, dec. 2012. doi: 10.1109/CODIS.2012.6422197.

[69] Francesco Salbalori. Proposal for a fault tolerant Hadoop Jobtracker, November 2008. URL http://sites.google.com/site/hadoopthesis/Home/FaultTolerantHadoop.pdf.

[70] Thomas Sandholm and Kevin Lai. Mapreduce optimization using regulated dynamic prioritization. In *SIGMETRICS '09: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, pages 299–310, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-511-6. URL http://doi.acm.org/10.1145/1555349.1555384.

[71] Lui Sha, Tarek Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K Mok. Real time scheduling theory: A historical perspective. *Real-time systems*, 28(2-3):101–155, 2004.

[72] Bikash Sharma, Victor Chudnovsky, Joseph L. Hellerstein, Rasekh Rifaat, and Chita R. Das. Modeling and synthesizing task placement constraints in Google compute clusters. In *2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 3:1–3:14, NY, USA, 2011. ACM. ISBN 978-1-4503-0976-9.

[73] Sun Microsystems, Inc. *Java Native Interface*. URL http://java.sun.com/docs/books/jni.

[74] David Talby and Dror G Feitelson. Supporting priorities and improving utilization of the ibm sp scheduler using slack-based backfilling. In *Parallel Processing, 1999. 13th International and 10th Symposium on Parallel and Distributed Processing, 1999. 1999 IPPS/SPDP. Proceedings*, pages 513–517. IEEE, 1999.

[75] Chunqiang Tang, Malgorzata Steinder, Michael Spreitzer, and Giovanni Pacifici. A scalable application placement controller

for enterprise data centers. In *Procs. of the 16th intl. conference on World Wide Web*, WWW '07, pages 331–340, NY, USA, 2007. ACM. ISBN 978-1-59593-654-7.

[76] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. In *Proceedings of the 2006 IEEE International Conference on Autonomic Computing*, pages 65–73, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 1-4244-0175-5. doi: 10.1109/ICAC.2006.1662383.

[77] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005.

[78] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2): 1626–1629, 2009.

[79] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. Data warehousing and analytics infrastructure at facebook. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 1013–1020, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0032-2.

[80] John Turek, Joel L Wolf, and Philip S Yu. Approximate algorithms scheduling parallelizable tasks. In *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pages 323–332. ACM, 1992.

[81] John Turek, Walter Ludwig, Joel L. Wolf, Lisa Fleischer, Prasoon Tiwari, Jason Glasgow, Uwe Schwiegelshohn, and Philip S. Yu. Scheduling parallelizable tasks to minimize average response time. In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '94, pages 200–209, New York, NY, USA, 1994. ACM. ISBN 0-89791-671-9. doi: 10.1145/181014.181331. URL http://doi.acm.org/10.1145/181014.181331.

[82] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. ARIA: Automatic Resource Inference and Allocation for Map-Reduce Environments. In *8th IEEE International Conference on Autonomic Computing*, Karlsruhe, Germany, June 2011.

[83] Tom White. *Hadoop: The Definitive Guide*. O'Reilly and Yahoo! Press, 2009. URL http://www.hadoopbook.com/.

[84] Adam Wierman and Misja Nuyens. Scheduling despite inexact job-size information. In *Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 25–36, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-005-0.

[85] Joel Wolf, Deepak Rajan, Kirsten Hildrum, Rohit Khandekar, Vibhore Kumar, Sujay Parekh, Kun-Lung Wu, and Andrey Balmin. FLEX: A Slot Allocation Scheduling Optimizer for MapReduce Workloads. In Indranil Gupta and Cecilia Mascolo, editors, *Middleware 2010*, volume 6452 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin / Heidelberg, 2010.

[86] Joel Wolf, Deepak Rajan, Kirsten Hildrum, Rohit Khandekar, Vibhore Kumar, Sujay Parekh, Kun-Lung Wu, and Andrey Balmin. Flex: A slot allocation scheduling optimizer for mapreduce workloads. In Indranil Gupta and Cecilia Mascolo, editors, *Middleware 2010*, volume 6452 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin / Heidelberg, 2010.

[87] Fatos Xhafa and Ajith Abraham. Computational models and heuristic methods for grid scheduling problems. *Future generation computer systems*, 26(4):608–621, 2010.

[88] Yahoo! Inc. Capacity scheduler. http://developer.yahoo.com/blogs/hadoop/posts/2011/02/capacity-scheduler/.

[89] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce performance in heterogeneous environments. In *8th USENIX Conference on Operating systems design and implementation*, pages 29–42, Berkeley, USA, 12/2008 2008. USENIX Association. URL http://dblp.uni-trier.de/db/conf/osdi/osdi2008.html.

[90] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Job scheduling for multi-user Map-Reduce clusters. Technical Report UCB/EECS-2009-55, 2009.

[91] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *5th European conference on Computer systems*, pages 265–278, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-577-2.

[92] Chen Zhang and H. De Sterck. Supporting multi-row distributed transactions with global snapshot isolation using bare-bones hbase. In *11th IEEE/ACM International Conference on Grid Computing*, pages 177 –184, oct. 2010. doi: 10.1109/GRID.2010.5697970.

[93] Chen Zhang and Hans De Sterck. Hbasesi: Multi-row distributed transactions with global strong snapshot isolation on clouds. *Scalable Computing: Practice and Experience*, pages –1–1, 2011.