Research on efficient resource utilization in data intensive distributed systems (データインテンシブ分散システムにおける高効率計算機資源利用に関する研究)

指導教員　喜連川 優
東京大学生産技術研究所

グロート　スヴェン
東京大学大学院
情報理工学系研究科
電子情報学専攻

平成 24 年 12 月 13 日提出

Abstract

In recent years there has been an unprecedented growth in the amount of data being gathered worldwide. Data may include business data, sensor data, web data, log data, and social data. Data volumes of terabytes are common, petabytes are not unusual anymore, and it is almost certainly not long before exabyte scale data becomes the norm.

Big data presents an opportunity for in-depth analysis. Trends and patterns can be derived from the data, enabling a wide variety of business and research opportunities. As a result, the ability to process large amounts of data—often too large to be stored and processed by traditional relational database systems—has become very important. Processing big data requires very large scale resources such as large clusters of commodity machines that allow the data to be processed in a distributed fashion. Google's MapReduce framework and its open source implementation Hadoop have emerged as the de facto standard data processing platform for such environments.

At the same time cloud computing has emerged, enabling users to dynamically provision computing resources and pay only for usage on a pay-as-you-go basis. Cloud computing enables anyone to gain access to large scale computing resources without the need to create their own infrastructure, drastically driving down the cost and making it possible for even individuals or small organizations to afford the resources necessary for processing big data. Cloud computing adoption has been driven by advances in virtualization and the wide-spread availability of high-bandwidth network connectivity.

When dealing with resources at this level of scale it becomes increasingly difficult to efficiently utilize all resources. With the billing model used by most cloud providers, provisioning a single node for ten hours costs the same as provisioning ten nodes for one hour, so it would be highly desirable if distributed applications could achieve a linear speed-up when more nodes or nodes with higher hardware specifications are provisioned. However, a number of factors limit the scalability of these applications so they cannot fully use the provisioned resources. In the cloud, inefficient resource utilization directly leads to higher costs for no additional benefit.

In this thesis, I investigate two major causes that contribute to the inability of data intensive distributed applications to efficiently utilize resources in cloud environments or other large scale distributed systems, and propose methods to mitigate their effects.

The first major cause of inefficient resource utilization is workload imbalance. This occurs when certain workers have a longer execution time than others, leading to stragglers: a handful of tasks that hold up an entire application. When a workload has stragglers, some computing resources provisioned for the workload may be idle, or partially idle, while waiting for the stragglers to finish. Because these resources are not being utilized they are unable to assist in speeding up the processing of the workload.

Workload imbalance and stragglers can have several causes such as data skew, processing skew and performance heterogeneity. Data skew occurs when the data to be processed is not divided evenly among the workers. Processing skew occurs when certain records in the data—even if they are not bigger than others—take more processing time. Performance heterogeneity can be caused by differences in the hardware between nodes, but also by environmental factors such as background processes active on certain machines, or interference between virtual machines when they are sharing the same physical host.

It is possible to mitigate data skew—and to a lesser extent, processing skew—by sampling the data beforehand, but these approaches cannot deal with performance heterogeneity. Performance heterogeneity is more complicated to account for beforehand, because precise knowledge about the hardware configuration is usually not available in the cloud and even identical instances can have very different performance. Although measurements can be used to establish the performance of each node, this can still not account for environmental factors that may change during the execution of the workload.

I propose a method called Dynamic Partition Assignment, which is able to dynamically adjust data distribution among workers whenever imbalance is detected. The workload is divided into many more partitions which are dynamically reassigned to workers that have already finished. Partitions are assigned to workers in groups that can be transferred in bulk in order to avoid the overhead of doing many small transfers. Because imbalance is lazily detected by monitoring the completion times of workers Dynamic Partition Assignment is able to handle data skew, processing skew and performance heterogeneity without any prior knowledge about the data or hardware environment.

Dynamic Partition Assignment was implemented in Jumbo, a MapReduce-style experimental data intensive distributed processing platform created for the purpose of experimenting with workload imbalance, and evaluated both for data skew and by running workloads on a heterogeneous environment. Dynamic Partition Assignment was able to successfully reduce the effects of stragglers, in some cases

improving processing times by 30 to 50%, and bringing the processing time to within 10% of the optimally balanced processing time.

The second major cause of inefficient resource utilization is I/O interference. When multiple applications are accessing the same resource simultaneously they can interfere with each other causing performance degradation. This is particularly a problem for I/O resources, such as disk storage, which are often shared between processes and can have a dramatic reduction in performance when under contention. For example, on a system with many CPU cores it can be desirable to run additional parallel processes to try to utilize these CPU cores, but when those processes contend for the same, more limited, I/O resources they are unable to utilize either the CPU or the I/O resources to their fullest effectiveness.

The nature of I/O interference means that it is very difficult to mitigate after it is detected to occur. Reassigning work often requires additional I/O to move data to new locations, exacerbating the problem. Therefore, it is desirable to be able to predict the effects of I/O interference before it actually occurs so that scheduling and resource provisioning can be adjusted accordingly.

I propose a cost model that is able to predict the effects of I/O interference when multiple MapReduce tasks running on the same node contend for that node's I/O resources. The model uses several workload parameters measured directly from running a subset of the workload, and a number of hardware parameters derived from micro-benchmarks, including hardware specific interference functions that describe how the storage devices behave under contention. These parameters and functions are used by an analytical model of MapReduce, which uses knowledge of MapReduce's processing flow and I/O patterns to predict the performance of the workload when using specified numbers of parallel processes.

The I/O interference model was evaluated against several representative MapReduce workloads and was able to predict their performance to within 5 to 10% even for highly I/O intensive workloads or workloads that use a combination of I/O and CPU intensive processing. The information provided by this model can be utilized, for example, to determine how many nodes to provision with how many CPUs, and how many tasks to run simultaneously on any given node. Additionally, this model can be used by a scheduler to decide how to place tasks in the cluster based on their expected effect on I/O performance.

Based on the I/O interference model, I have developed Mariom, a framework and toolset for gathering hardware and workload parameters and making predictions based on those parameters using the I/O interference model. This framework can be integrated into scheduling or provisioning systems to provide the ability to more accurately reason about situations where I/O interference occurs.

Improving resource utilization in the cloud helps to reduce application execution time, and reduces costs for both cloud providers and users. In this thesis, I address two aspects of this problem: I propose Dynamic Partition Assignment to

mitigate the effect of stragglers and improve workload balancing, and I propose a cost model that addresses the problem of I/O interference. While both of these are targeted at MapReduce, the methods used in this thesis are not specific to MapReduce and can be applied to other data intensive applications in the cloud.

Acknowledgments

I would like to express my sincere gratitude to my supervisor at the University of Tokyo, Professor Masaru Kitsuregawa, for his guidance and support for the duration of this work. I also wish to thank him for allowing me the opportunity to come to Japan to conduct research at his laboratory.

I want to thank Associate Professor Miyuki Nakano, Associate Professor Kazuo Goda, and Dr. Daisaku Yokoyama for their invaluable help, advice and assistance in my research. The many meetings and countless hours of discussion were immeasurably helpful in keeping me on track and helping to focus my work. Without them, this thesis would not exist. I would also like to thank the members of my Ph.D. committee: Professor Jun Adachi, Professor Syuichi Sakai, Associate Professor Masahiro Goshima, Associate Professor Kenjiro Taura, and Associate Professor Masashi Toyoda.

I also want to thank Professor Harry A.G. Wijshoff at Leiden University in the Netherlands, who encouraged me to seek out this opportunity to come to Japan as a Ph.D. student.

I want to thank my family, in particular my parents, who have been separated from me for these long years but who nonetheless offered endless encouragement and support, even when things did not proceed according to plan. I also want to thank the people who created Skype for making this separation much more bearable than it otherwise would have been.

Finally, I want to thank my good friend Nora Leonard for helping to keep me sane these past few years, for stimulating my creativity, and for making me realize how lucky I am that I did not have to write this thesis on a typewriter.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The amount of data worldwide has seen explosive growth in recent years. Companies, institutions and individuals produce ever more data, including business data, sensor data, web data, log data, social data from sites such as Facebook or Twitter, and more. This growth is fueled by an increasing amount of data sources, as data is produced by ubiquitous mobile devices, aerial sensory technologies, software logs, cameras, RFID readers, wireless sensor networks, and so forth. According to the 2011 IDC Digital Universe Study [GR11], the total amount of data created and replicated in 2010 exceeded 1.2 zettabytes, with a predicted growth to 35 zettabytes by 2020.

Examples of big data are numerous:

- Google reported it processes 20 petabytes of data each day [DG08] in order to improve the quality of its web search engine.
- In 2010, the four main detectors of the Large Hadron Collider (LHC) at CERN in Switzerland produced 13 petabytes of data [Bru11].
- The Sloan Digital Sky Survey [YAAJ+07] captures data at a rate of 200 gigabytes per day, producing over 140 terabytes in total. Its successor, the Large Synoptic Survey Telescope [Swe06] is projected to produce that much information every five days.

Due to the proliferation of big data, analyzing and processing very large quantities of data has become a prime concern. Big data affects not just traditional fields such as Internet search, finance and business informatics, but also scientists in the fields of meteorology, genomics, connectomics, complex physics simulations,

and biological and environmental research. In all cases, important information can be derived by studying trends and correlations in very large data sets.

Data analysis is traditionally accomplished using relational database systems, with the use of parallel databases to cope with larger volumes and long duration analysis jobs. However, one of the defining characteristics of big data is that it is too large to be processed in these kinds of systems, so alternative methods need to be used.

A landmark in the area of distributed data processing was the development of the MapReduce [DG04] programming model by Google. MapReduce enables scalable, fault-tolerant processing of terabytes or petabytes of data on large clusters of commodity hardware, which allows for a substantial reduction in the cost of data processing. MapReduce also provides a simple programming model, allowing developers to easily create large scale distributed data analysis applications without having to worry about the complexities normally associated with distributed and parallel programming.

Google's MapReduce model was replicated by Apache Hadoop [Apaa], providing an open source implementation that was adopted by Yahoo!, Amazon, Facebook, and many others. Through Hadoop, MapReduce has become the de facto standard solution for large scale data processing, and it remains a target of much active research.

Starting from 2007, cloud computing has begun to emerge as an alternative to traditional cluster computing and grid computing. Cloud computing provides on-demand access to very large scale computing resources using a pay-as-you-go model, eliminating the need for organizations and individuals to maintain their own infrastructure. Key features of the cloud include the ability to rapidly provision compute and storage resources, the ability to share resources between multiple tenants, and the ability to elastically respond to changes in demand. Cloud computing has rapidly grown in importance, and this growth is expected to continue; the 2011 IDC Digital Universe Study estimates that by 2015, as much as 10% of information (around 0.8 zettabyte) will be retained in the cloud, and in total almost 20% will be "touched" by cloud computing service providers somewhere during its journey from originator to disposal.

Thanks to cloud computing it is now within the reach of nearly anyone to gain access to large scale computing resources without the need to invest in establishing their own infrastructure, enabling even individuals or small companies to process very large data sets. This brings with it a number of challenges for both the cloud provider and the users.

Cloud systems provide users with access to a large number of resources. Compute nodes—often provided as virtual machines—encompass CPU, memory, and local disk resources. Typically, a persistent large scale storage system is also provided. Finally, linking all these together are network links between the various

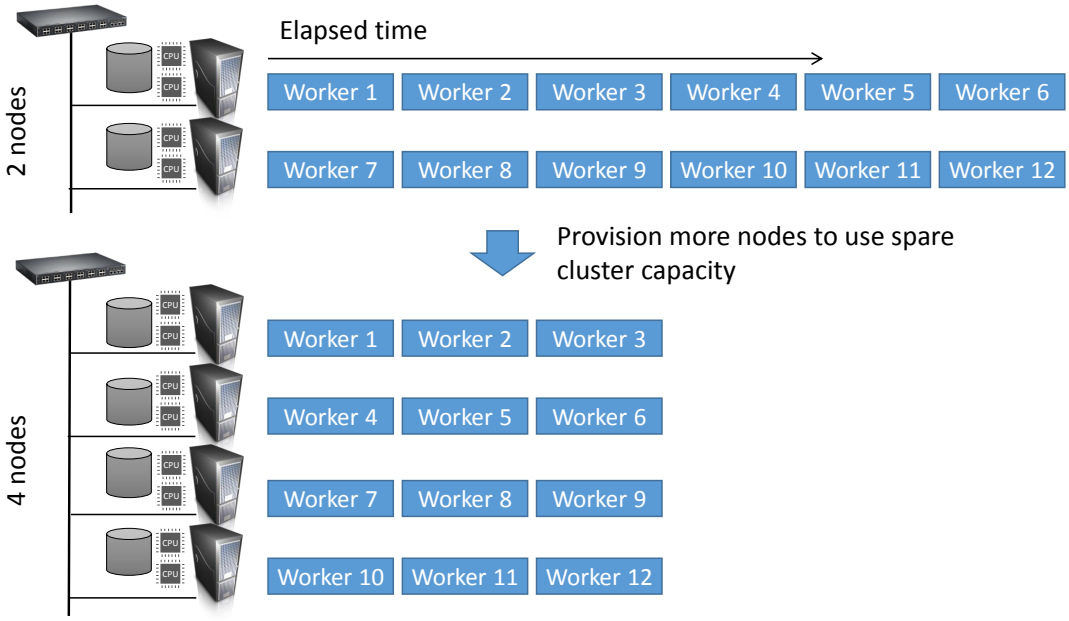


Figure 1.1: The ideal situation when more nodes are provisioned

nodes.

Being able to utilize all these resources at scale is a challenging issue, and one that has received a lot of attention in the research community. For providers it's important that they are able to provide their customers with the best service, and service as many customers as possible within the infrastructure they have at their disposal. For users the primary concern is doing the most work with the least resources, as that translates directly into lower costs.

The pricing model of most cloud environments means that the issue of scale-out is very important. With most cloud providers, it costs the same amount of money to use one node for ten hours as it would cost to use ten nodes for one hour. If applications can support linear scale-out, this means an increase in speed is attainable at no extra cost.

How an application scales is affected by its structure. Applications in the cloud often consist of many individual worker processes that are scheduled to run on the available resources. Figure 1.1 shows what would ideally happen with those workers if the number of nodes is doubled: the workers are spread evenly over the extra machines, so doubling the number of nodes leads to double the performance, with no additional cost thanks to the cloud pricing model.

However, in actuality this rarely ever happens. Some of the problems are highlighted in Figure 1.2:

- Because of skew in the input data or the amount of processing to be done

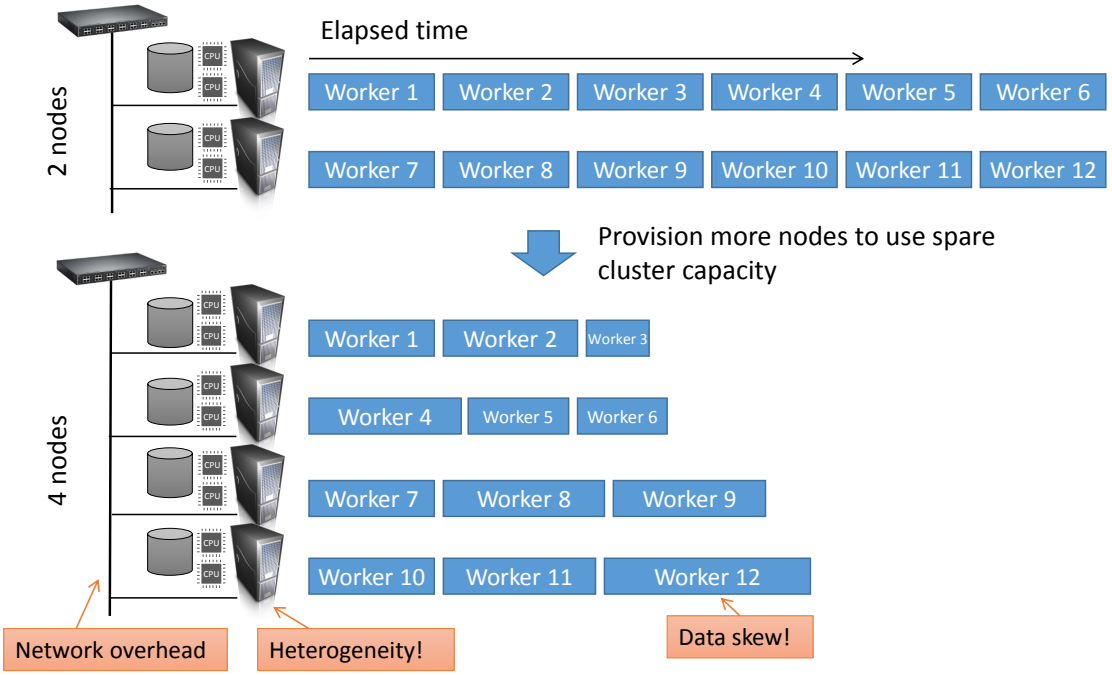


Figure 1.2: The actual situation when more nodes are provisioned

per worker, not all workers may in fact be executing for the same length of time. As a result, when the workers are distributed over more nodes, not all the nodes are working for the same amount of time. The overall processing time of the workload is decided by those workers that finish last, and it is possible for a small number of workers, called stragglers, to dominate the execution time of the workload. While those stragglers are finishing some other resources may be idle, which means they are being wasted while they are still being paid for.

- The nodes may not all be identical; in fact, they are very likely to be heterogeneous. Even if the nodes appear to be homogeneous, their performance is often not identical, particularly if the nodes are virtual machine that may be placed on one physical host together with other virtual machines. For example, on Amazon's cloud it is well known that even seemingly identical compute instances can offer wildly varying performance [SDQR10, LYKZ10]. For this reason, tasks that executed in a certain amount of time on one node may take a completely different amount of time on another. This means that some workers may become slower, and it also causes further variance between the tasks leading to stragglers.

- When more nodes are added to a distributed processing job, this may entail network communication overhead as the workers communicate and transfer data between nodes. Depending on the speed of the network (which is also known to suffer large fluctuations from instance to instance on typical cloud platforms), the overhead of this additional communication can be substantial. This will reduce the performance of the workers and therefore increase the amount of time it takes to run the entire workload.

Because of these issues, it is likely that while adding additional nodes to a distributed processing application will improve performance, that improvement is likely to be less than linear. As a result, you are using N times the number of nodes for more than $\frac{1}{N}$th of the original time, leading to an increase in cost. Thanks to stragglers some of these nodes are also likely to be idle for some of the time, yet you are still paying for them for the entire length of the job. Additionally, it is very difficult to predict how the performance of a data processing workload will change if more nodes are added and consequently what the additional cost will be, making it very difficulty for the user to decide if the trade-off between higher costs and shorter completion times is worth it.

The issue of scalability does not just apply to provisioning multiple nodes, but can also affect worker execution on a single node. Many modern systems have multiple CPUs, and for example Amazon offers a number of compute instances where the virtual machine has access to more than one of the CPUs of the underlying physical host.

In the case where the workers are not naturally capable of exploiting parallelism with multiple CPUs, the straightforward approach taken by many current large scale data processing frameworks is to simply schedule more tasks to run simultaneously on the same node. Figure 1.3 shows the ideal situation here: these systems each have two CPU cores, so the number of parallel workers on each node is increased from one to two. If linear scale-up is possible, performance will double by doing this.

However, the reality is not so simple, as Figure 1.4 shows. Not only can workers on the same system suffer from the same kind of skew issues as with multiple nodes, they are also subject to resource contention.

While the number of CPU cores on a system keeps increasing, and those CPUs are able to execute independently to a large degree, other subsystems are not keeping pace. Simultaneously executed workers on a single node need to access a number of shared resources, including memory, storage, and in some cases CPU cache in the case of a multi-core processor. Of these resources, the storage devices are the most subject to severe and highly variable interference. Since data intensive applications are typically dominated by disk I/O this can have a significant impact on the overall performance of parallel workers sharing a single storage resource.

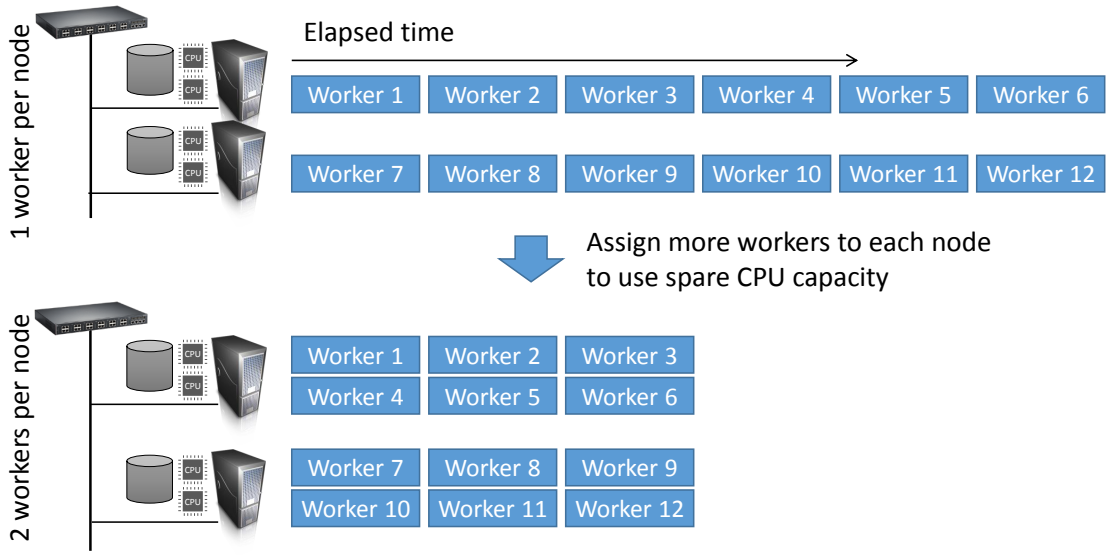


Figure 1.3: The ideal situation when more tasks are added to each node to exploit multiple CPU cores on each node

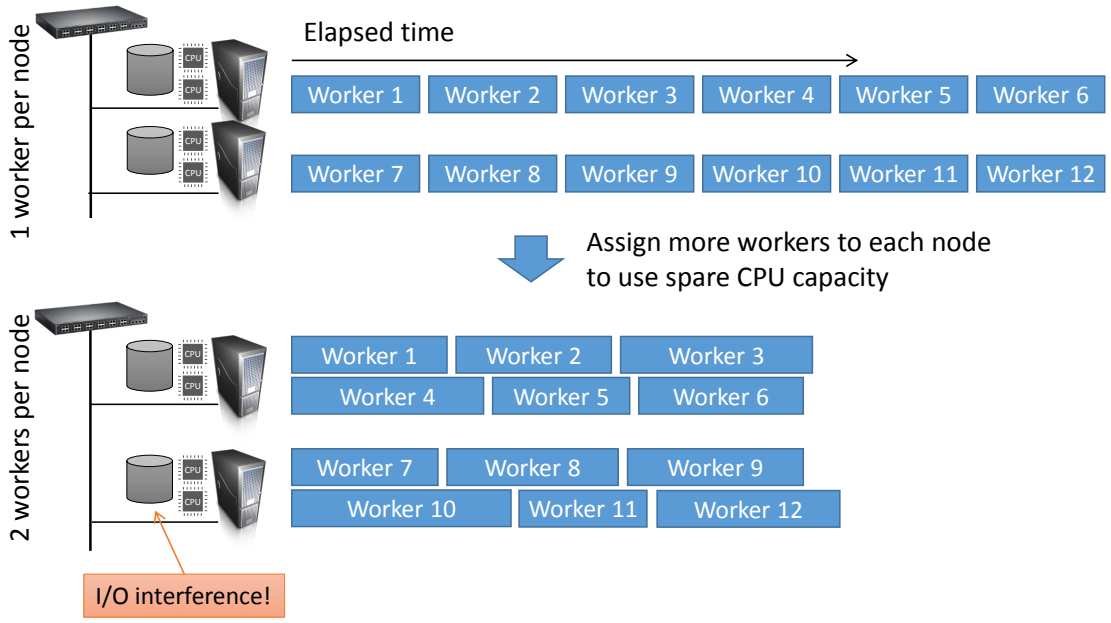


Figure 1.4: The actual situation when more tasks are added to each node to exploit multiple CPU cores on each node

Storage systems can range from simple local hard disks to large arrays of disks or a networked large-scale persistent storage system. In most cases however, these will be presented as a single resource to the applications on the system. Whereas two applications running on a multi-core system can both utilize their own CPU core, they must share this I/O resource even if it is comprised of many individual disks internally.

Traditional spinning platter disks are particularly sensitive to I/O interference, as executing multiple workloads increases the number of seek operations and therefore increases I/O latency and decreases performance. While smart read-ahead caching algorithms and the use of SSD drives (which perform much better for random access workloads than traditional hard disks) can alleviate some of the problems, the core issue of sharing a single resource between multiple concurrent applications remains. This issue is further compounded when multiple virtual machines on the same physical host are using the same physical storage device, or when multiple nodes in a cloud environment are utilizing a shared persistent storage system (for example, a storage system located on a SAN or NAS).

As a consequence of I/O interference, the execution time of the various workers will increase. If the workers are entirely I/O bound, running multiple workers on the same system will have no performance benefits at all. At best, it will take the same amount of time to execute as just having a single worker at a time, and at worst, thanks to the non-linear performance degradation of hard disks, performance will actually decrease.

Due to the complex nature of storage systems, it is even more difficult to predict the effect of multiple workers per node than with multiple nodes. It is therefore nearly impossible for a user to decide how many tasks to run per node, how to mix workloads of different I/O and CPU intensity, or even what kind of compute instances (with what number of CPU cores) to use in the first place.

It is clear that it is highly desirable to improve the resource utilization of data intensive applications, leading to improved scalability in both these scenarios. Not only does this translate to improved execution time and lower costs for end users, cloud providers are likewise able to utilize their infrastructure more efficiently and would be able to service more customers with the same hardware.

1.2 Research goals

The goal of the research in this Ph.D. thesis is to improve resource utilization for data intensive applications running in the cloud or cloud-like environments by mitigating some of the factors indicated in Figure 1.2 and Figure 1.4.

In this area, this work makes the following contributions:

- Improved understanding of the behavior of data intensive distributed sys-

tems. The issues outlined in Section 1.1 are examined in more detail for their causes and their impact on performance.

- Enable workload balancing of data intensive applications. In particular, I propose *Dynamic Partition Assignment* as a method to dynamically redistribute work between workers when imbalance is detected at runtime. This method was implemented on top of an experimental system for data intensive processing, and found to be an effective method to mitigate the effect of stragglers and improve utilization of resource that would otherwise have been idle for part of the execution time.

- Allow mitigation of the effects of I/O interference. Unlike data skew or hardware heterogeneity, which can be dealt with upon its occurrence during the execution of a job, I/O interference has a more complex effect on performance. Although it is possible to detect hotspots, any attempt to improve performance after a hotspot occurs often means discarding some of the work already done (as there is no way to move in-progress workers without killing them in current data intensive systems) and/or incurring additional I/O to relocate data from the hotspot to the resources that the work has been reassigned to. As such, it is preferable to be able to use performance prediction to prevent hotspots from forming and make decisions about worker execution based on those predictions. I propose a performance model that is able to predict the effects of I/O interference on data intensive applications, allowing the application or user to make more informed decisions about per-node parallelism.

- Based on the I/O interference model, I created a tool to allow this model to be applied to any workload.

1.3 Thesis outline

The remainder of this Ph.D. thesis is structured as follows:

- Chapter 2 provides an introduction to cloud computing, including its history, primary features, and applications, and how this research pertains to the features of cloud computing.

- Chapter 3 provides an overview of data intensive distributed systems on the cloud, focusing on MapReduce and its most popular implementation, Hadoop. Some alternatives to MapReduce are also discussed.

- Chapter 4 covers runtime workload balancing and the issues pertaining to stragglers caused by skew or heterogeneity. The chapter discusses *Jumbo*, an experimental distributed data processing system that was created to investigate these issues, and introduces *Dynamic Partition Assignment*, my method for runtime workload balancing based on dynamic reassignment of work during execution, which is implemented and evaluated using Jumbo.

- Chapter 5 discusses the issue of I/O interference, in particular as it pertains to per-node parallelism. An I/O performance prediction model is introduced, which is evaluated for prediction accuracy using MapReduce on a cluster environment with several representative real-world workloads.

- Chapter 6 introduces Mariom, a framework and toolset developed to allow the I/O interference model to be applied in production environments.

- Chapter 7 concludes the thesis and discusses future work to be done based on this research.

Chapter 2

Cloud computing

2.1 Overview

Although the term cloud computing has many conflicting definitions, it is most commonly defines as the use of computing resources that are provided over a network, most commonly the Internet. The name is thought to be derived from the stylized cloud shape that is often used to represent networks or the Internet in system diagrams. With cloud computing, the data, software and computation belonging to a user are entrusted to a remote service, often offered by a third party.

Cloud computing enables users to avoid expensive and complex provisioning of infrastructure and software, instead allowing them to utilize the infrastructure, platform and software offered by commercial or open source cloud initiatives. Cloud computing comprises many services and applications, as shown in Figure 2.1. Cloud computing is often referred to as a form of utility computing, comparing the model used by cloud computing to that of utilities such as the electricity grid. Utility computing means that the costs of the service are based on usage, rather than having to buy infrastructure up front, just like the electricity grid charges users for the amount of electricity consumed.

Amazon was one of the major drivers of cloud computing, by first modernizing their internal data centers to use a cloud-like model. After discovering that the new architecture offered significant benefits and improvements in internal efficiency, Amazon created a new product to provide cloud services to external customers in the form of Amazon Web Services [Amaa].

Since 2007, cloud computing has rapidly gained attention and importance, and has become a new industry buzzword. Figure 2.2 informally shows this increase in interest by using Google Trends to compare the search volume of the term cloud computing with grid computing, which it has rapidly overtaken.

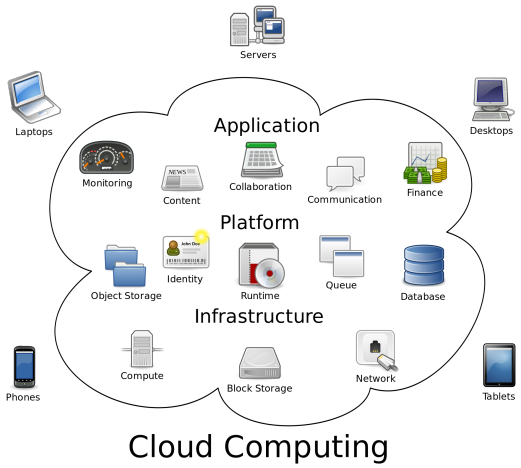


Figure 2.1: A logical diagram of cloud computing

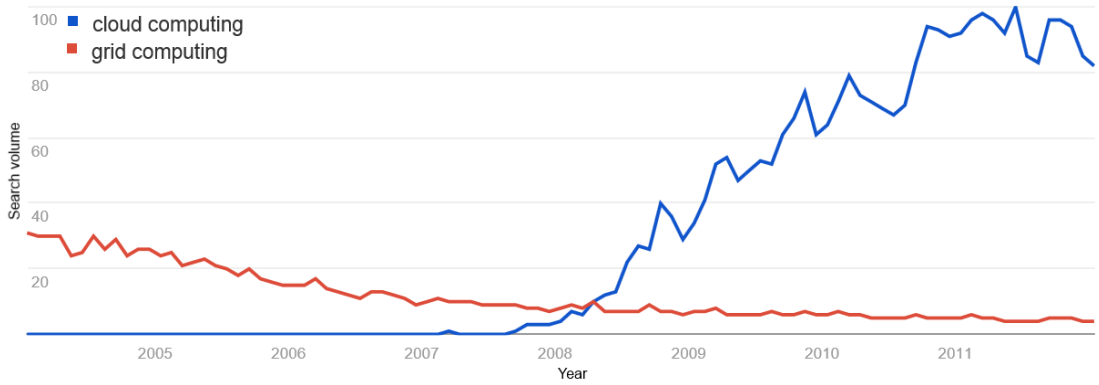


Figure 2.2: Search trend for cloud computing according to Google Trends

2.1.1 Types of cloud computing

Cloud computing can take many forms, but the most common service models are outlined below:

Infrastructure as a Service (IaaS) This is the simplest model of cloud computing. The cloud provider offers the user computers as physical machines or more typically through the use of virtual machines, as well as other hardware resources. Virtual machines are executed in hypervisors such as Xen [BDF+03] or KVM [KKL+07]. Other resources may include raw (block) and file based storage, firewalls, load balancers, virtual LANs, libraries of virtual machine images, and software bundles.

Cloud users must install their own operating system, or use one of the provided images, to utilize the hardware. Therefore, the user can run any kind of software configuration desired on any of the machines. This does however mean that cloud users are responsible for maintaining the operating systems and software themselves. This service is usually charged as a utility computing service, with pricing depending on the amount of time you are consuming the resources.

Examples of Infrastructure as a Service include Amazon EC2 [Ama10], the Rackspace Open Cloud [Rac] and Windows Azure Virtual Machines [Micc].

Platform as a Service (PaaS) Cloud providers provide a computing platform, usually including an operating system, APIs, database and web server. End users develop software solutions that run on top of this platform. This avoids the complexity of having to manage software configuration manually. Additionally, these platforms often offer built-in support for scaling based on demand.

Examples of Platform as a Service include Amazon Elastic Beanstalk [Amab], Google App Engine [Gooa], and Windows Azure Compute [Micc].

Software as a Service (SaaS) In this model, cloud providers provide entire software applications. This software is hosted in the cloud by the provider, with installation and configuration managed entirely by the provider and not the user. The user accesses the application through a web browser or a light-weight client application. Cloud applications are similar in many ways to traditional web applications, but they often offer far greater scalability, utilizing load balancers to elastically scale to utilize more machines during times of high demand.

Examples of Software as a Service include Google Apps [Goob] and Microsoft Office 365 [Mica].

2.1.2 Cloud deployment models

Clouds can be deployed in several different ways:

Public Cloud Cloud applications, storage, and other resources are made available for public use by a cloud provider such as Amazon, Microsoft, or Google.

Community Cloud A cloud that is shared by several organizations with common concerns, managed either internally or by a third party.

Hybrid Cloud A combination of multiple separate clouds (public, community, or private) that remain unique but are linked together.

Private Cloud Cloud infrastructure intended for use by a single organization, managed internally or by a third party.

2.1.3 Features of cloud computing

Cloud computing has several key features. Because cloud computing is a relatively new field and there is still some confusion over what the term does and does not entail, it is possible to find different definitions of what a cloud is and what its features are. The NIST Definition of Cloud Computing [MG11] defines the following essential characteristics, which are widely considered to be the key features of a cloud computing platform:

On-demand self service Users can provision required resources such as computing time and storage automatically without human interaction with the service provider.

Broad network access The cloud's capabilities are available over a network through standard mechanisms that promote the use of heterogeneous thin and thick client platforms.

Resource pooling The provider's resources are pooled to support multiple tenants simultaneous, with different physical and virtual resources automatically assigned to different users based on demand without their control or knowledge.

Rapid elasticity The ability to quickly provision and release resources on demand, in order to respond to rapidly changing increases and decreases of application demand. From the user's perspective, resources appear limitless and can be appropriated at any time.

Measured service Resource usage is controlled and optimized automatically by the use of metering capability through some level of abstraction appropriate to the service (e.g. storage, network bandwidth, processing).

Elasticity and multi-tenancy are particular features of interest in the cloud. Elasticity allows the user to quickly provision seemingly limitless resources based on demand without the need to commit beforehand to a desired capacity. This allows e.g. start-ups to respond to unexpectedly rapid growth without needing the massive investment of acquiring and installing the infrastructure to respond to such growth. Multi-tenancy means that workloads from multiple organizations and individuals are multiplexed on the same infrastructure, enabling better utilization.

Cloud computing's adoption has been largely driven by advances in virtualization. The use of virtual machines allows for rapid configuration of software

environments irregardless of the actual underlying hardware configuration, and virtual machines can be assigned and reassigned to different physical hosts according to demand and resource availability. Virtual machines of different cloud users can co-exist on the same physical host without the users being aware of each other (although in practice, hosting multiple virtual machines on a single physical host has performance and security implications).

The cloud still offers many challenges for researchers and providers to overcome. Armbrust et al. [AFG+10] outline challenges and opportunities related to availability, lock-in, confidentiality and auditability, data transfer bottlenecks, performance unpredictability, scalable storage, bugs in large distributed systems, quick scaling, reputation fate sharing, and software licensing.

2.2 Resource utilization in the cloud

The issues of resource utilization discussed in Section 1.1 are relevant to the cloud. Particularly the features of resource provisioning, elasticity, multi-tenancy relate to the topic of resource utilization.

With cloud computing, the user is able to provision resources automatically, but it is often not clear in advance what resources are required for a particular workload. In the case of repeatedly executed workload, the user may gain some intuitive understanding of the behavior of the workload on different types and amounts of available cloud resources, but for new workloads or variable workloads this is not so easily answered.

Cloud providers typically offer a wide variety of compute instances, differing in the speed and number of CPUs available to the virtual machine, the type of local storage system used (e.g. single hard disk, disk array, SSD storage), whether the virtual machine may be sharing physical resources with other virtual machines (possibly belonging to different users), the amount of RAM, network bandwidth, etc. In addition, the user must decide how many instances of each type to provision.

In the ideal case, more nodes means faster execution, but issues of heterogeneity, performance unpredictability, network overhead, and data skew mean that the actual benefit of utilizing more instances can be less than expected, leading to a higher cost per work unit. These issues also mean that not all the provisioned resources may be optimally used for the duration of the application. Workload skew may mean that some of the provisioned resources are (partially) idle and therefore do no contribute to the performance during those periods, but still contribute to cost. Provisioning larger or higher performance instances is similarly not always able to yield a proportional benefit.

Because of these factors, it can be very difficult for a user to translate their performance requirements or objectives into concrete resource specifications for the

cloud. There have been several works that attempt to bridge this gap, which mostly focus on VM allocation [HDB11, VCC11a, FBK+12, WBPR12] and determining good configuration parameters [KPP09, JCR11, HDB11]. Some more recent work also considers shared resources such as network or data storage [JBC+12], which is especially relevant in multi-tenant scenarios. Other approaches consider the provider side of things, because it can be equally difficult for a provider to determine how to optimally service resource requests [RBG12].

Resource provisioning is complicated further because performance in the cloud is not always predictable, and known to vary even among seemingly identical instances [SDQR10, LYKZ10]. There have been attempts to address this by extending resource provisioning to include requirement specifications for things such as network performance rather than just the number and type of VMs in an attempt to make the performance more predictable [GAW09, GLW+10, BCKR11, SSGW11]. Others try to explicitly exploit this variance to improve application performance [FJV+12].

Accurate provisioning based on application requirements also requires the ability to understand and predict application performance. There are a number of approaches towards estimating performance: some are based on simulation [Apad, WBPG09], while others use information based on workload statistics derived from debug execution [GCF+10, MBG10] or profiling sample data [TC11, HDB11]. Most of these approaches still have limited accuracy, especially when it comes to I/O performance. Cloud platforms run a wide array of heterogeneous workloads which further complicates this issue [RTG+12].

Related to provisioning is elasticity, which means that it is not always necessary to determine the optimal resource allocation beforehand, since it is possible to dynamically acquire or release resources during execution based on observed performance. This suffers from many of the same problems as provisioning, as it can be difficult to accurately estimate the impact of changing the resources at runtime, and therefore to decide when to acquire or release resources, and which ones.

Exploiting elasticity is also further complicated when workloads are statically divided into tasks, as it is not always possible to preempt those tasks [ADR+12]. Some approaches for improving workload elasticity depend on the characteristics of certain workloads [ZBSS+10, AAK+11, CZB11], but these characteristics may not generally apply.

It is therefore clear that it can be very difficult to decide, for either the user or the provider, how to optimally provision resources and to ensure that those resources that are provisioned are utilized fully. Their is a very active interest in improving this situation, and the approaches proposed in this thesis similarly aim to improve provisioning and elasticity by mitigating common causes of inefficient resource utilization.

Chapter 3

Data intensive processing on the cloud

3.1 Introduction

As applications need to deal with increasingly large amounts of data, traditional relational database systems are no longer able to effeciently handle data at this scale. In addition to not being able to cope with the immense size of the data (often terabytes or even petabytes, with exabytes likely not far way), much of the data being dealt with is unstructured or only partially structured. For example, a web search engine needs to build an index by processing vast amounts of HTML data.

Initially, this lead to the creation of many custom distributed applications, but this is a very complex and error prone process and most people are not experts in the intricacies of distributed and parallel programming. Debugging distributed applications is even more difficult, so finding and fixing problems with these custom solutions was often a costly and time-consuming process.

Additionally, when a large amount of computing resources are used for processing big data, failures are common. This is especially the case because of a trend in the early 2000s to use large clusters of relatively cheap commodity machines for this purpose, which was found to be more cost effective than using traditional high-performance computing solutions, despite the fact that this hardware would often fail. Not only individual disks and computers are subject to failure, if network switches fail entire racks of machines could become unavailable. Even in large public clouds, where hardware is often of higher quality than those cheap commodity clusters, the sheer scale of the systems involved means that failures are common and must be expected and dealt with.

As a result of how common it is for hardware to fail, it is of paramount im-

portance that the application is able to withstand failures without data loss, and with only minimal loss of work. In order to ensure this, fault tolerant large scale data storage is required. And because big data workloads often take many hours to complete, it is not acceptable if a hardware failure leads to the re-execution of the entire workload. Instead, the workload should be divided in such a way that hardware failures lead to only minimal re-execution, and should not lead to complete job failure even if an entire rack is lost or any of the data storage becomes corrupt.

Writing such highly fault tolerant systems makes creating distributed data processing applications even more complicated than it already was, and it is unreasonable to expect most programmers to be able to deal with this. It is no surprise therefore that generic frameworks for distributed processing have become popular. These frameworks provide built-in fault tolerance, as well as an easy programming interface for developers to create distributed applications without worrying about the difficulties normally associated with this kind of development. Instead, they only need to worry about solving their particular problem within the framework's programming environment.

With the rise of cloud computing, distributed data intensive processing is used by more and more people, so many of the frameworks that were originally used only on large in-house clusters are now deployed on the cloud. In this chapter, I will discuss these frameworks, focusing primarily on MapReduce, which has become the de facto standard for large scale data processing.

3.2 Distributed data storage

In order to be able to provide reliable data processing, one first needs to have reliable data storage. Distributed data storage should be resilient against disk failures, network failures, or data corruption, ensuring data integrity and availability under the most common failure conditions. In addition, these storage systems need to be able to handle the large volumes of often unstructured data that are common in these kinds of big data processing applications.

3.2.1 Google File System

Probably the most influential recent work in distributed data storage is the Google File System [GGL03] (GFS). GFS was designed specifically to handle systems where failures are common, and to handle very large amounts of data typically stored in very large files.

The Google File System's architecture was based on the following assumptions, which were derived from observations about workloads on Google's production

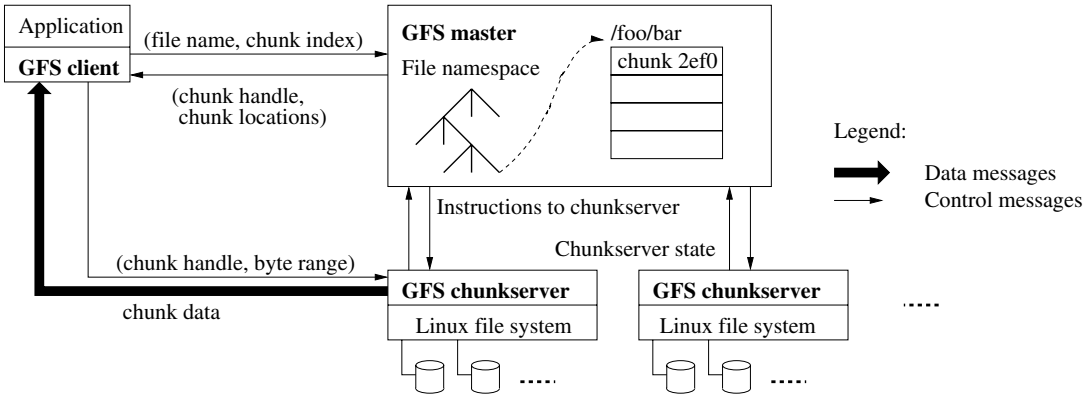


Figure 3.1: GFS Architecture

clusters:

- The system is built from inexpensive commodity hardware that often fails.
- The system stores a modest number of large files.
- Workloads primarily consist of large streaming reads and small random reads.
- Workloads often do large, sequential writes that append data to a file. Writes to existing offsets in a file are not as common.
- The system must support well-defined semantics for concurrent writers to a single file.
- High sustained bandwidth is more important than low latency.

Figure 3.1 shows the basic architecture of GFS. File system metadata is stored by a single *GFS master*, and file data is divided into large *chunks* that are stored on a *GFS chunkserver*.

The master keeps track of the file system namespace, and the list of chunks that make up each file. In addition, it stores the list of chunk servers for each chunk. All file system metadata is kept in memory on the master for fast retrieval, and any mutations to the namespace are stored in an operation log, which is persisted on disk and replicated across multiple machines. When the master restarts, it recovers file system information by replaying the operation log. In order to minimize restart time, regular checkpoints are written which allow for faster recovery. These checkpoints can be created online without halting further file system mutations while the checkpoint is being made.

The master does not keep a persistent record of chunk locations. Instead, chunk servers report which chunks they have upon first contact with the master, from which it constructs the list.

GFS uses the Chubby lock service [Bur06] to select a master.

Chunks store file system data, and are stored on the chunkservers. Chunks are replicated on multiple machines. Chunks are typically very large, often 64MB or more, which is much larger than previous distributed file systems. Using such large chunks reduces the clients' need to interact with the master, it can reduce network overhead by keeping a persistent TCP connection with a chunkserver when a client performs many operations on a single chunk, and it reduces the metadata stored on the master. Chunks use checksumming to detect data corruption. If a chunk replica is lost or detected to be corrupt, the chunk is automatically re-replicated.

When a client reads data, it resolves the file name and offset on the master to a chunk ID, and then resolves a list of chunk servers for that chunk. It selects a chunk server to read from (preferably a local or rack-local chunk, if possible), and contacts it directly. Data transmission does not flow through the master so it does not become a bottleneck. For writing, a client similarly writes data directly to a single chunk server, which then forwards the data to the replicas to utilize the maximum full-duplex bandwidth at each chunkserver.

Replica placement is an important consideration in the fault tolerance and performance of a system. GFS uses a replica placement policy that ensures that replicas are always placed on multiple racks so that rack failure does not lead to data loss.

GFS supports multiple concurrent writers by providing an atomic append operation. The client simply provides the data to write, and GFS decides at what offset to write it. This way, multiple writers do not hinder each other.

GFS has inspired many similar systems. Most notably, the Hadoop Distributed File System [Apaa] (HDFS) provides an open source implementation of the GFS model, which has gained widespread adoption. The Kosmos Distributed Filesystem [Kos] is an alternative that offers strict POSIX compliance as well as Hadoop compatibility.

3.2.2 Cloud storage

Cloud systems often offer persistent storage with the goals of scalability, high availability, low latency and low cost.

Amazon S3 [Amad] is the storage service offered on Amazon's cloud service, allowing the storage of arbitrary objects up to 5 terabytes organized into buckets. Windows Azure [Micc] offers blob storage as well as relational storage and non-SQL others. Other cloud environments often also offer their own persistent storage. For

commercial cloud platforms, the precise implementation details of these storage systems are usually not revealed.

3.3 Distributed data processing

The need to process large quantities of often unstructured data has led to the creation of a number of frameworks for that purpose. Traditional and parallel relational database systems are unable to cope with the size and nature of the data, and often also do not offer the reliability guarantees required for such large scale systems.

A distributed data processing framework offers a programming model to the user that allows for the easy creation of distributed applications without needing to worry about the minutiae of the distributed operation. Often this requires breaking down an application into several steps or operators that can be easily parallelized by the framework. The framework then provides an execution environment to run programs written using their programming model in a scalable, fault tolerant way.

3.3.1 MapReduce

The MapReduce [DG10, DG08, DG04] framework was first proposed by Google in 2004, and since then has become the de facto standard for large scale data processing in the cloud. MapReduce has been deployed in a wide variety of environments, including most cloud environments. For example, Amazon offers Elastic MapReduce [Amac], and Microsoft has begun offering Hadoop on its Windows Azure cloud solution through Windows Azure HDInsight [Micb].

MapReduce programming model

MapReduce provides a simplified programming model, hiding the details of parallelization, fault tolerance, data distribution and load balancing in a library. The abstraction for the programming model is based on the *map* and *reduce* primitives present in Lisp and many other functional languages.

A MapReduce computation takes a set of input key/value pairs, and produces a set of output key/value pairs. The user writes a *Map* and *Reduce* function that operates on these pairs.

Map This function processes a single input key/value pair and produces a set of *intermediate* key/value pairs. The MapReduce framework groups together all intermediate values with the same intermediate key I and passes them to the *Reduce* function.

Reduce This function processes a key I and the set of associated values for that key. These values are merged together to form a new, possibly smaller, set of values. Many *Reduce* functions produce just zero or one output value, though this is not always the case.

The canonical example MapReduce program used is the problem of counting the number of occurrences of each word in a large collection of documents. A pseudo-code version of a program solving that problem using MapReduce is given below:

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

The `map` function takes each word in the document, and emits an intermediate key/value pair using the word as the key and the value 1. The `reduce` function sums all the counts for a particular word.

Although this pseudo-code uses only string inputs and outputs, conceptually they can be of any type. Google's MapReduce implementation passes strings to the user function, leaving it up to the function to convert them to other types.

MapReduce execution

MapReduce executes by automatically partitioning the input data into M splits and invoking the *Map* function for each of them, allowing the splits to be processed in parallel by multiple machines. Intermediate key/values pairs are partitioned by key into R pieces using a partitioning function, with each *Reduce* invocation processing one of the partitions.

Figure 3.2 shows the general flow of execution in a MapReduce program. The numbers correspond to the following steps:

1. The MapReduce framework splits the input files into large pieces (e.g. 64MB per piece). It then starts many copies of the program on a cluster of machines.

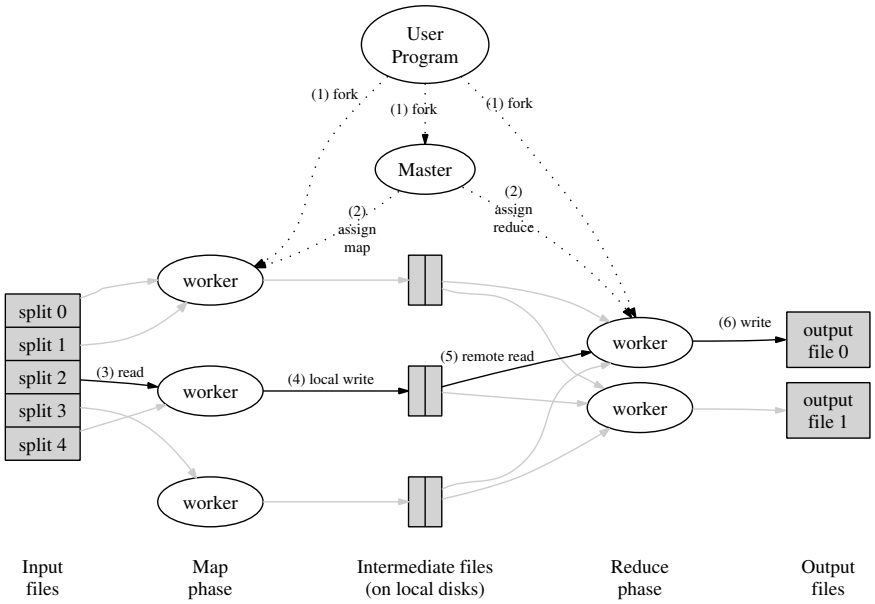


Figure 3.2: MapReduce execution overview

2. One of the program copies acts as the master, while the others are workers that are assigned work by the master. There are M map and R reduce tasks to assign.

3. A worker running a map task reads the input split, and parses the key/value pairs from the input and passes each pair to the map function. The intermediate key/value pairs are buffered in memory.

4. Periodically, the buffers are written to disk, partitioned into R groups by the partitioning function. The location of these partitions is passed to the master, which forwards it to the reduce workers.

5. Reduce workers use remote procedure calls to read the intermediate data from the local disks of map workers. When all intermediate data is read, it is sorted by key so that all identical keys are grouped together. An external sort is used if the data does not fit in memory.

6. The reduce worker iterates over the data for each unique key, passing the key and its set of values to the reduce function. The output of the reduce function is appended to the final output.

7. The master notifies the user program of the job completion.

MapReduce uses the Google File System described in Section 3.2.1 for input and output storage (intermediate data is stored on the local disk for each worker). GFS divides files into chunks which are replicated (typically three copies), and these chunks provide natural splits for the map tasks. The master uses information about the location of each chunk when scheduling map tasks, and attempts to schedule a map task on a machine containing a replica of that chunk. If that fails, it attempts to schedule the map task on a machine near a machine containing a replica of that chunk (e.g. a machine in the same rack).

Fault tolerance is handled by the master. Each worker sends a periodic heartbeat to the master, and failure to receive a response from a worker for a certain time indicates failure of the worker. Any map tasks executed by that worker must be re-executed. This includes completed map tasks because their output is stored on the local disk, and may therefore not be available anymore if a machine running a worker fails. This is not necessary for reduce tasks because their output is stored on the distributed file system.

It is possible to handle master failure by having it write periodic checkpoints, but Google's implementation of MapReduce does not do this.

In the case of failures it is possible that a task gets executed multiple times. In this case, MapReduce relies on atomic commits of task completions to ensure that only one copy of the output is retained. Reduce tasks write to private output files that are moved to the final output only once the task successfully commits.

MapReduce handles stragglers by scheduling backup tasks of the remaining in-progress tasks when the job gets close to completion. This means that tasks running on cluster nodes that are performing badly (e.g. because of CPU, memory, local disk, or network interference) are simultaneously executed on different machines, using whichever task finishes first as the final output.

In some cases, there is a lot of repetition in keys in the output of a single map class. In the case of a reducer that significantly reduces the set of values for a particular key, it can be beneficial to run the reducer locally to reduce intermediate data size, and therefore network traffic. This type of usage of a reduce function is called a *Combiner* function. It requires that the reduce function is commutative and associative. Note that it is not required for the combiner and reduce function to be the same for a given workload.

MapReduce criticisms

Since its original inception, MapReduce has become the de facto standard for large scale data processing. As a result, it is being used for a wide variety of workloads far beyond what it was originally designed for. Some of these workloads are not well-suited for the MapReduce paradigm, particularly if these workloads are the kind of workloads traditionally done by relational databases. Many of these workloads

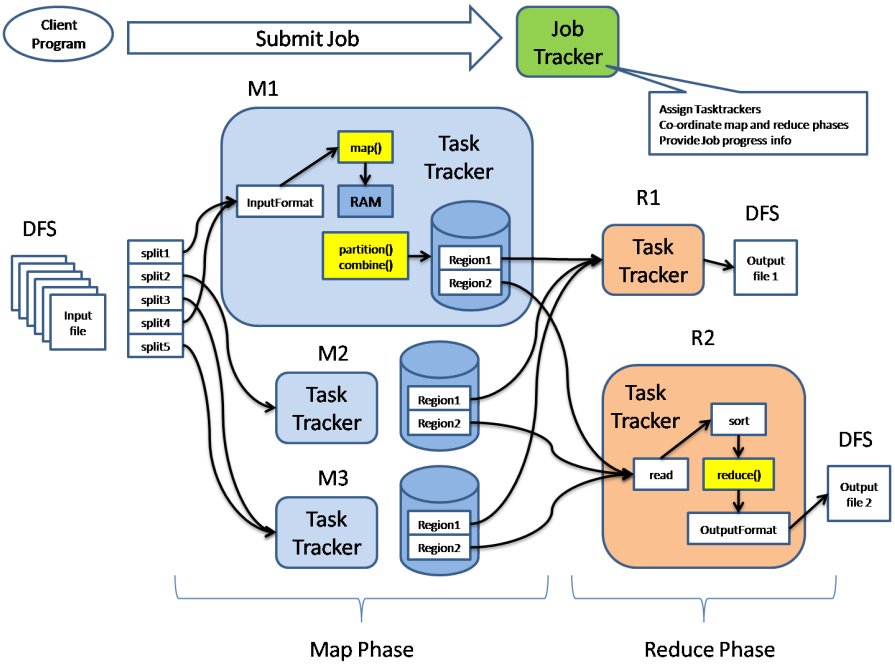


Figure 3.3: Hadoop MapReduce architecture

can benefit from the ability to utilize indices or native join support, which most implementations of MapReduce do not offer.

As a result, MapReduce has been criticized for adding unnecessary overhead to many computation jobs, with parallel database systems often outperforming common MapReduce implementations such as Hadoop by a factor of ten for typical relational workloads [PPR+09].

However, this criticism is not necessarily fair, as it depends primarily on the MapReduce implementation (most of which are based directly on Google's original description so do not offer improvements such as indices that these kinds of workloads have been optimized for. Works such as Hadoop++ [DQRJ+10, JQRD11] and HadoopDB [ABPA+09] add relational database features like indexing to MapReduce, alleviating most of these criticisms.

Hadoop

Hadoop [Apaa] provides an open source implementation of the MapReduce programming model and execution framework, largely following the description presented above.

Hadoop uses a centralized *Jobtracker* that is responsible for handling submission and execution of jobs, including scheduling of workers and tasks. Each node in the cluster runs a *Tasktracker* daemon that communicates with the Job-

tracker, and is responsible for executing workers on that node. The architecture of Hadoop's MapReduce implementation is shown in Figure 3.3. Note that this is based on the current stable branch of Hadoop (Hadoop 1.0, based on the Hadoop 0.20 codebase).

The architecture of the current alpha version of Hadoop (Hadoop 2.0, based on the Hadoop 0.23 codebase) is very different. This architecture, called Hadoop YARN, uses a centralized resource scheduler called the *ResourceManager*, but leaves the details of job progress and fault tolerance to application specific *ApplicationMasters*. Hadoop YARN is therefore closer to the original Google MapReduce implementation than the current stable versions of Hadoop.

Hadoop typically uses the Hadoop Distributed File System (HDFS), which is an open source GFS-like file system, for storage of input and output files. However, Hadoop supports using other storage mechanisms in place of HDFS as long as there exists an implementation of Hadoop's abstract Filesystem interface for them.

Hadoop was originally created by Doug Cutting while he was working at Yahoo! as infrastructure for the Nutch open source web search engine [KCSR04]. Hadoop has gained wide spread adoption and is used among others by Yahoo!, Amazon, Facebook, Twitter, and LinkedIn. Hadoop is supported on many cloud environments and is the most common MapReduce implementation used across the industry, and therefore the de facto standard distributed data processing application currently in use.

MapReduce ecosystem

Because of MapReduce's popularity, a large amount of related projects have sprung up. Many of these use Hadoop as the underlying MapReduce implementation.

- *HBase* [Apab] is an open source, non-relational distributed database modeled after Google's Bigtable [CDG+08]. It uses HDFS to store the database, and supports large scale processing jobs through Hadoop MapReduce using HBase both as a source and sink.

- *Hive* [TSJ+09] is a data warehousing solution for Hadoop.

- *ZooKeeper* [HKJR10] is a high performance distributed coordination service that can be used to manage Hadoop deployments.

- *Mahout* [Apac] is a library of machine learning algorithms for MapReduce, implemented on Hadoop.

- *Pig* [ORS+08] provides a procedural data processing language that gets compiled down into Hadoop MapReduce programs.

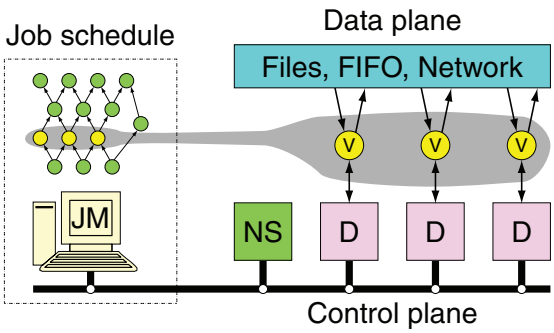


Figure 3.4: Dryad system organization

- *Jaql* [BEG+11] provides a declarative scripting language for data processing based on Javascript that is also executed using Hadoop MapReduce.

This is just a small sampling of the many projects related to MapReduce that have sprung up as its popularity has increased over the last several years.

3.3.2 Alternatives to MapReduce

Although MapReduce is by far the most prolific framework for large scale data processing, some alternative approaches have been proposed. Despite the advantages that some of these approaches have over MapReduce, they have never gained any real traction in the community.

Microsoft Dryad

Dryad [IBY+07] represents a more flexible approach to parallel and distributed programming. Whereas MapReduce uses a rigid structure of a map function connected to a reduce function, Dryad represents jobs as a directed acyclic graph, where vertices represent programs and edges represent communication channels between those programs. Vertex programs are automatically scheduled to run on a cluster of machines, and there may be far more vertices in a graph than there are nodes in the cluster. Vertex programs can perform almost any operation, and can be structured into complex graphs allowing for a far more flexible, but also more complicated, approach to structuring parallel workloads.

Channels between vertex programs are used to transport a set of items. Channels are an abstraction for which Dryad provides several implementations, including shared memory (for processes on the same machine), TCP pipes, or temporary files persisted on a file system. The vertex programs themselves are agnostic of the underlying channel implementation.

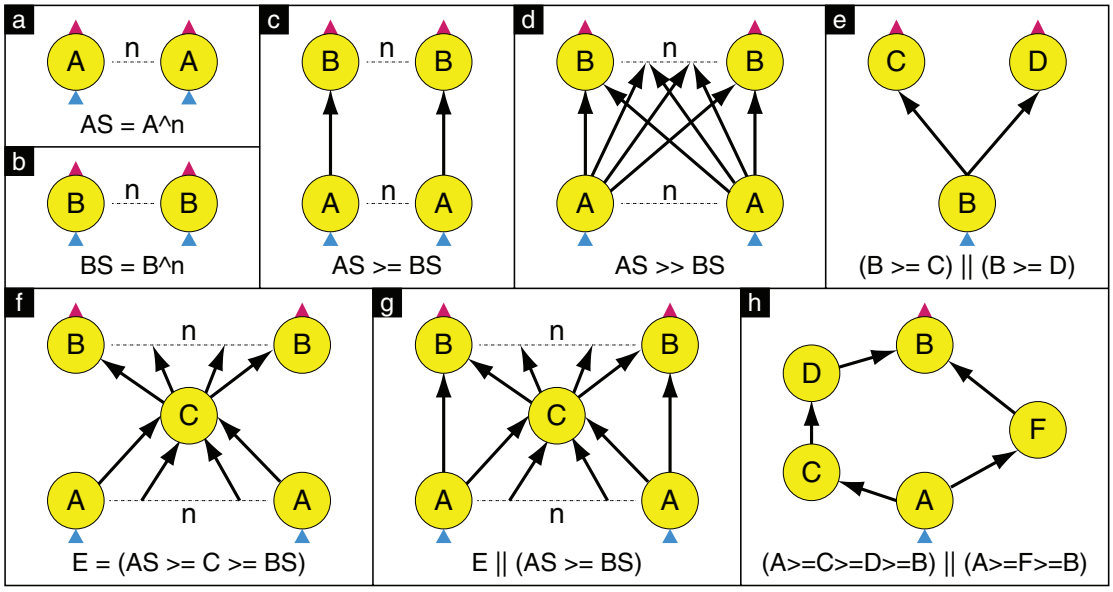


Figure 3.5: Dryad graph description language operators

Figure 3.4 shows the system organization of Dryad. The job manager (JM) consults with a name server (NS) to enumerate the list of available computers and discover their topology. A simple daemon (D) is running on each computer that is responsible for executing vertex programs (V) on behalf of the job manager.

Dryad's power comes from the flexibility in creating graphs. Dryad uses a simple language for describing common communication paradigms embedded into C++ using a combination of method calls and operator overloading. Figure 3.5 shows the operators of the graph description language. Boxes (a) and (b) show the cloning of vertices, communication between vertices is shown in (c) using point-wise composition and in (d) using complete bipartite composition, and (e) demonstrates a merge. Boxes (f), (g) and (h) show more complex patterns of composition.

A simpler method to construct Dryad jobs is offered by DryadLINQ [YIF+08], which uses the Language INtegrated Query (LINQ) functionality of Microsoft .Net. With DryadLINQ, LINQ queries are automatically compiled down to Dryad job graphs.

Nephele

Nephele [WK09] is a platform for distributed programming that bears a strong resemblance to Microsoft's Dryad. Nephele similarly specifies a job as a directed acyclic graph. However, unlike Dryad where the graph's vertices represent exactly one program instance and graph modifications must be manually applied in order

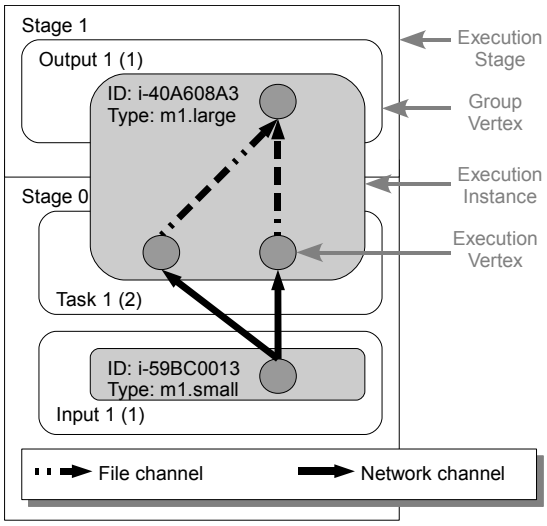


Figure 3.6: Example of a Nephele execution graph

to create parallelization, Nephele allows the user to specify for each vertex whether it is suitable for parallelization, in which case the task is automatically divided into subtasks by the Nephele engine. Nephele also allows annotations to specify for example the node instance type that is best suited for a specific task.

Like with Dryad, Nephele's job structure allows each vertex to have multiple inputs and outputs, making it much easier to represent certain operators than with MapReduce.

When executing a job, the job structure is transformed into a job execution graph, which contains concrete information about subtasks and communication channels between tasks. Execution graphs are divided into stages for scheduling, where each stage can only be scheduled when the preceding stage is entirely complete, and Nephele ensures all instances required for a stage are allocated before it begins processing. An example of a Nephele execution graph is given in Figure 3.6.

PACTS [BEH+10] is a programming model for Nephele that extends the MapReduce model with additional operators and output contracts for task programs, enabling better optimization opportunities than solutions like Pig that target MapReduce directly.

SCOPE

SCOPE [CJkL+08, ZBW+12] is a parallel processing system that combines benefits from both parallel databases and MapReduce-style systems.

Similar to parallel databases, the system has a SQL-like declarative scripting language with no explicit parallelism, while being amenable to efficient parallel

execution on large clusters. An optimizer is responsible for converting scripts into efficient execution plans for the distributed computation engine. A physical execution plan consists of a directed acyclic graph of vertices. Execution of the plan is orchestrated by a job manager that schedules execution on available machines and provides fault tolerance and recovery, much like MapReduce systems.

SCOPE executes on top of the Cosmos distributed execution engine, a system developed for internal use by Microsoft with similar design goals as MapReduce. Cosmos also forms the basis for Dryad.

3.3.3 Extensions to MapReduce

Besides complete alternatives to MapReduce, MapReduce has also been extended and adapted for many purposes.

Indexing and relational features

Based on the criticisms in [PPR+09], MapReduce has been extended to incorporate features like indexing and join support.

HadoopDB [ABPA+09] implements the MapReduce programming model on top of a distributed relational database engine, providing intrinsic support for many features Hadoop and other implementations of MapReduce lack.

Hadoop++ [DQRJ+10] introduces join-oriented indices into Hadoop DFS files, enabled MapReduce programs to transparently utilize these indices without needing to be modified. This approach is extended further to utilize Hadoop DFS's replication to introduce different indices in each replica, so that the optimal one can be chosen based on the job's data access pattern [JQRD11].

An alternative approach is to translate MapReduce programs into SQL to execute against a relational database [IZ10].

Cheetah [Che10] and ES2 [CCG+11] propose the utilization of column-based stores for MapReduce. Similarly, another work [FPST11] creates a separate physical file for each attribute.

Iterative processing

A number of works extend MapReduce with support for iterative computations.

HaLoop [BHBE10] extends the MapReduce programming model with support for iterative computations, and increases their efficiency by making the task scheduler loop aware and adding various caching mechanisms.

The iMapReduce framework [ZGGW11a] supports iterative computation and avoids re-shuffling static data without modifying the scheduler.

Twister [ELZ+10] utilizes a light-weight MapReduce runtime with all operations performed in memory.

PrIter [ZGGW11b] enables fast iterative computation by providing the support of prioritized iteration. Instead of performing computations on all data records without discrimination, PrIter prioritizes the computations that help convergence the most, so that the convergence speed of iterative process is significantly improved.

Chapter 4

Runtime workload balancing for data intensive applications

4.1 Introduction

Cloud applications operating on big data are usually divided into a number of workers, which are executed in a distributed fashion on the nodes of the cluster. In the case of MapReduce these workers execute map and reduce tasks. There are typically a large number of workers; far more than the cluster can execute simultaneously. In a typical MapReduce workload the number of map tasks may be orders of magnitude larger than the number of nodes, and while the number of reduce tasks is usually lower it will still usually be equal to the number of nodes or a small multiple of that. Each worker is responsible for processing a part of the job's data. Map tasks process a part of the input data (an input split), and reduce tasks process a partition of the intermediate data.

If workers don't take the same amount of time to execute, they are said to be skewed. There are a number of reasons why skew can occur between workers:

- *Data skew* means that not all tasks process the same amount of data. Those tasks that process more input data will likely take longer to execute. Data skew can occur due to the properties of the input data, but also for example due to a poor choice of partitioning function.

- *Processing skew* occurs when not all records in the data take the same amount of time to process. Even if the tasks process the same amount of data and records, there can still be a large discrepancy in their execution times.

- *Performance heterogeneity* can mean that even when tasks the same amount of work, they may not take the same amount of time. This may occur

because the computing resources in the cluster are actually heterogeneous, with some nodes having faster CPUs, more network bandwidth, more RAM, or more and faster disks than others. These nodes will be able to process data faster than the others, and run the same tasks faster. Even identical nodes can suffer from performance heterogeneity because of interference from background tasks, network congestion, or in the case of a virtual machine running in the cloud, interference from other virtual machines sharing the same host. This problem can also occur across machines, as a worker that needs to transfer data from a machine on a different rack may have less bandwidth available than a worker that only needs to transfer data from the same rack or even the local host.

The end result of these factors is that there can be a large variation between the execution time of the workers. When this occurs, some workers may hold up the execution time of the entire job, either because other workers cannot proceed until they are finished, or because they are simply the last workers in the job. These *stragglers* can dominate the execution time of a workload, and this is a known problem for MapReduce and similar data processing frameworks.

Whenever there are stragglers, it means that there are resources in the cluster that are not being utilized while they are waiting for the stragglers to complete. However, since these resources are still provisioned for your application, under the cloud's typical payment model you are still paying for them even though they are not actively being used. Dynamically releasing unneeded resources and re-acquiring them later when needed again is difficult due to the typical setup time of compute instances, and even when currently unused the resources may still be necessary for completion of the job (for example, they may hold part of the intermediate data).

Although it is possible to utilize the idle resources for other workloads if there are multiple workloads (possibly belonging to multiple users) in the system, it can still be problematic to meet performance guarantees or QoS specifications for a particular job when workload imbalance occurs.

Google's MapReduce implementation [DG04] tries to deal with the problem of stragglers by scheduling backup tasks; basically, if a job is nearing its end and certain tasks are taking longer than expected, backup instances of those same tasks will be executed by different workers in the hope that they can finish them faster. Hadoop, the most commonly used MapReduce implementation, uses the same strategy and calls it *speculative execution*.

Speculative execution can be an effective way to combat task skew caused by performance heterogeneity, as it is indeed possible for other machines to finish the same tasks faster. But in the case of data skew or processing skew the backup task will take the same amount of time as the primary task, and since it was

started later it will not finish before the primary. In this case, the backup task does completely redundant work which will be discarded once the primary finishes. Even if the backup task does improve performance, duplicate work was performed by the primary which will be discarded once the backup finishes. In the worst case, a backup task may add additional network overhead or interfere with other tasks on the node running the speculative task, actually diminishing overall performance.

There are some ways to try and prevent stragglers before executing the workload. Data skew can be determined by sampling the input beforehand and adjusting e.g. the partitioning function to compensate. This is much more difficult for processing skew because the characteristics that make a record more computationally intensive to process may be difficult to determine automatically. In any case, this requires an additional sampling step that adds to the overall execution time.

Performance heterogeneity can be addressed by utilizing knowledge about the physical system configuration, but this is often not available for cloud users, and even compute instances with identical specifications can perform very differently [SDQR10]. An alternative is to utilize performance measurements, but this can only account for static differences between the nodes (for example, faster CPU or more memory). It cannot account for performance differences caused by temporary environmental factors such as interference from other tasks or network congestion.

It is therefore preferable to be able to mitigate workload imbalance when it occurs. That way, all factors can be accounted for and no a priori knowledge about the data or systems is necessary.

In this chapter, I describe issues in workload balancing as they pertain to MapReduce, and propose *Dynamic Partition Assignment*, a method of dynamically redistributing workload in response to imbalance in the reduce phase while maintaining the efficiency of large transfers.

4.2 Related work

The performance of MapReduce remains an area of very active research interest [JOSW10, PPR+09, SAD+10].

The problem of stragglers was first noted in the original paper on MapReduce by Dean and Ghemawat [DG04], which proposes the use of backup tasks (called speculative execution in Hadoop) to run additional copies of stragglers so that those copies may be able to finish faster. Zaharia et al. [ZKJ+08] improve the use of speculative tasks specifically for heterogeneous environments. Ananthanarayanan et al. [AKG+10] developed a method to improve the decision process of whether to restart a task or use speculative execution. Guo et al. [GF12] look at speculative execution and resource stealing particularly in the situation where

there are fewer tasks than cluster resources. Yang et al. [YYTM10] propose a technique for handling data skew in the map phase only.

Many of the existing methods methods that attempt to deal with skew in MapReduce such as[KBHR10, ORS+08, RLC+12, TSJ+09] rely on executing a job twice, at least partially. The first execution samples the data to determine its distribution. Because partitioning actually happens on the output of the map functions, such sampling techniques must necessarily execute the map function to sample the intermediate data rather than the map functions. These methods rely on well-known sampling techniques studied in the database community [MRL99, MBLL07, HBK05].

Techniques that rely on sampling are capable of dealing with data skew, but incur overhead before a job to perform the sampling. Additionally, unless the job's computations are fully performed on all sampled records, they cannot identify processing skew (and even if it can be identified, processing skew is likely to depend on the value rather than the key, making it difficult to adjust partitioning to alleviate processing skew). Unless knowledge of the hardware environment is incorporated, these methods can also not deal with performance heterogeneity.

Sailfish [RRS+12] avoids the sampling step by recording statistics during actual job execution and maintaining an index for each partition that allows it to split them further across multiple workers. The number of workers per partition is decided at runtime based on partition size. However, this means that reduce workers cannot start until the map workers have finished, and it is also not able to deal with processing skew or performance heterogeneity.

Vernica et al. [VBBE12] use an alternative approach, where the number of partitions is fixed but the partitioning function is constructed dynamically so that the partitions will be balanced. This approach can only be used for data skew.

Ramakrisnan et al. [RSU12] address the problem of large keys by using a method to split large keys into multiple partitions which can be done if the reduce function is associative and commutative (the same as the requirements for a combiner function). Progressive sampling is used to identify the large keys. This approach then uses key packing to distribute work across reducers, treating partitioning as a bin packing problem rather than more traditional partitioning approaches. Since this is based on the size of the keys, it is also only applicable for data skew.

SkewTune [KBHR12] uses an approach to mitigating skew that uses a similar lazy skew detection method as my approach and dynamically redistributes the workload by creating new tasks to process the additional partitions. The drawback of this approach is that it must read and re-partition an entire input partition at the moment skew is detected, which is potentially an expensive and I/O intensive operation.

Workload balancing and skew mitigation in general are widely studied in the

context of database systems [WDJ91, DNS+92, SN95, PI+96, TK99, GTOK02, XK09]. Most of these efforts are related to joins and parallel aggregation. Some of these mechanisms can be adapted for use on MapReduce. For example, the Pig system includes a SkewedJoin [GNC+09] based on [DNS+92].

FLUX [SHB04] uses an approach quite similar to my own to divide query operators into mini-partitions. However, this approach is aimed principally at fault tolerance rather than skew mitigation, and suffers from well-known issues with creating many small tasks [KBHR10], which my approach avoids.

Load balancing in the cloud has also been extensively studied. Randles et al. [RLTB10] provide an overview of workload balancing mechanisms employed in the cloud.

4.3 Workload imbalance issues in MapReduce

4.3.1 Map stage

A typical MapReduce job consists of a large number of map tasks. These map tasks are scheduled to run on nodes in a greedy fashion. The scheduler prefers to run a task on the node where its input block is located. However, if this can't be done a map task will always be scheduled on a node if there is capacity available and map tasks remaining. Until the job runs out of map tasks to schedule, stragglers are not an issue.

There are some special circumstances that can cause execution time skew between map tasks. Locality may play a role, as tasks that must read their input data remotely may perform slower than those that can read from local disk. Data placement and the scheduler's ability to schedule map tasks on the same node as their input data can therefore affect individual task performance. Data skew, particularly skew in the size of a map task's output data, can cause large variance. Intermediate key/value pairs are collected in a buffer, which is periodically sorted and spilled to disk. If a map task's output does not fit in the buffer, multiple spills are necessary which must be merged to create the map task's final output. Tasks that need to perform such a merge can be significantly slower than tasks that don't.

Additionally, even if the tasks are all identical in terms of the amount of processing performed, I have found interference from running multiple tasks in parallel on the same node can cause their execution times to vary wildly.

Once there are no more tasks to schedule for the job, the remaining map tasks may become stragglers and hold up execution of the reduce tasks. Because the MapReduce framework relies on sorting to group the records by key, a reduce task cannot begin to process its records until all intermediate data has been transferred

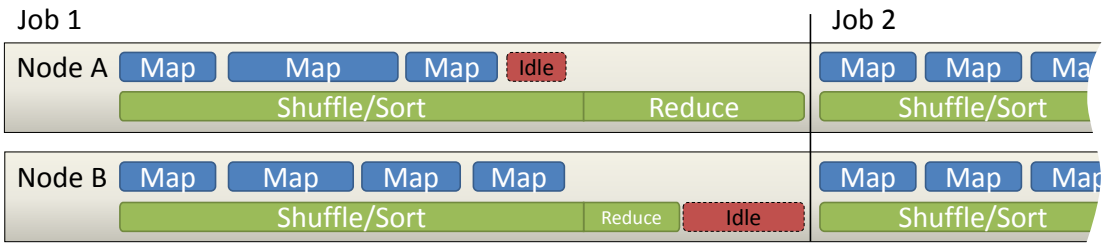


Figure 4.1: Workload imbalance from stragglers in MapReduce

and sorted, which means all map tasks must finish first. While reduce tasks can be scheduled to run while there are still unfinished map tasks for the job, they can only shuffle and sort, but not yet begin processing the reduce function. As a result, some of the resources of the cluster are likely idle while the job waits for the remaining map tasks. This situation is depicted in Figure 4.1, which shows that node A is (partially) idle while waiting for the final map task to finish on node B. The reduce tasks on both nodes cannot start reduce function processing until that final task is completed.

While stragglers may be an issue for map tasks, there are several factors that mitigate their effect:

- There are typically a very large number of map tasks (often orders of magnitude more than the cluster can execute simultaneously). This means that performance heterogeneity is automatically addressed, as those machines that are able to process tasks faster will simply process more tasks.
- The individual tasks are relatively short compared to the overall duration of the workload, so the amount of idle time at the end of the map phase caused by stragglers is also relatively short. If the task skew is extremely severe, this may not be the case.
- Because map tasks process fixed size input splits, it is common for most of the map tasks to process very similar amounts of data, so data skew should be limited. Hadoop also sorts map tasks by their data size, executing the smallest tasks last, again mitigating the effect of stragglers.

4.3.2 Reduce stage

A MapReduce job typically has fewer reduce tasks than map tasks. Typically, the number of reducers is chosen such that the cluster can complete all tasks in one or two waves. This is done because of the large overhead of running additional tasks, which is caused by framework overhead for running each task, the fact that

background shuffle and sort can only be performed for the first wave of tasks, and the overhead of performing additional network transfers.

As a result, reduce tasks are fewer in number and often longer in duration relative to the overall length of the workload. This means that the effect of stragglers can be much more severe.

In order to divide data amongst the reduce tasks, MapReduce partitions intermediate data by key using a user-specified partitioning function (by default, the function `hash(key)` is used). Depending on the nature of the data and the quality of the partitioning function, severe data skew can occur. The intermediate data may naturally be clustered around a few keys or even just a single key. Alternatively, the partitioning function may have a lot of collisions, assigning a large number of keys to a small subset of the partitions.

If a straggler occurs in the reduce phase, some of the cluster's resources will be idle. There are no further tasks for this workload to schedule, and the workload will not be finished until the last reduce task completes. The slowest task therefore determines the overall execution time of the workload.

Many complex MapReduce applications consist of multiple MapReduce jobs, often in a sequence where the output of one job forms the input of another. In this situation the second job cannot start until the first job is completely finished, because it needs all of the first job's output files to be available in order to compute its input splits. This situation is also shown in Figure 4.1, where the reduce task on node A is a straggler causing node B to go unutilized until that straggler finishes. Job 2 in the figure depends on the input data of job 1, so it cannot start until job 1 is completed.

4.3.3 Example: Parallel FP Growth

Throughout this chapter, I will use a particular application to demonstrate the problems with workload imbalance, and to test my proposed solution in the later sections. The application used is Parallel FP Growth [LWZ+08], a MapReduce version of the popular FP Growth [HPY00] algorithm for frequent pattern mining.

In frequent pattern mining, the input is a database of transactions, where each transaction consists of a set of items. The goal is to find out which items frequently (more often than a user-specified *minimum support*) occur together in the same transaction.

Figure 4.2 shows the steps of the FP Growth algorithm, which are as follows:

1. Count the number of occurrences of each item in the transaction database.
2. For each transaction, remove infrequent items (those that have fewer occurrences than the minimum support, which is 3 in the example in Figure 4.2) and sort the remaining items in order of descending frequency.

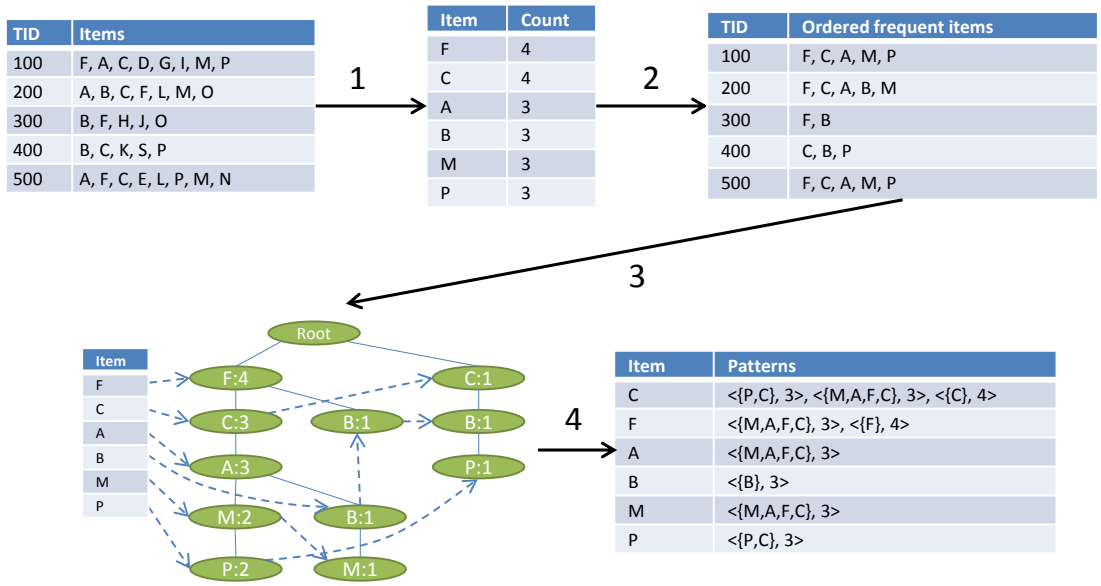


Figure 4.2: Processing workflow of the FP Growth algorithm

3. Construct an FP Tree representing the database. Each node is labeled with an item and frequency count, except for the root of the tree which is a special node. Starting at the root of the tree, for each item in a transaction it will check if a child node already exists for the item, and if so it is used and its count incremented; if not a new child node is created with a count of 1. Simultaneously, a header table is maintained linking the nodes that are created for each item.

4. Mine the FP Tree for frequent patterns. Starting at the least frequent item in the header table, follow the links to each node and if that node's count is larger than the minimum support, follow the parent of each node back to the root to find the patterns containing that item and emit them as part of the output. This is repeated for each item in the header table.

The output of the algorithm is a list of frequent patterns with their number of occurrences.

The version of FP Growth used by Parallel FP Growth is a slight modification of this algorithm. Instead of using a minimum support to filter the output, the algorithm reports the top-k frequent patterns for each item. To facilitate this, the patterns for each item are stored in a size-bounded priority queue.

FP Growth is not a simple algorithm to parallelize. Parallel FP Growth uses the following approach:

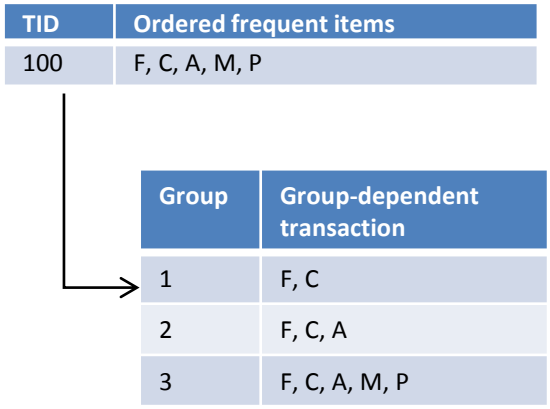


Figure 4.3: Creating group-dependent transactions from a transaction

1. Use a parallel counting job to create the frequency list for the items. This is a very straightforward operation supported by MapReduce.

2. The frequency list is divided into groups, where each group contains N items. N has to be set small enough that the FP tree for each group is likely to fit in to memory.

3. The transactions in the database are filtered into group-dependent transactions. For each group for which there is an item in the transaction, a group-dependent transaction is created containing those items and any more frequent items in the group. Figure 4.3 shows how group-dependent transactions are generated for a transaction using the frequency list from Figure 4.2 divided into three groups.

4. For each group, the FP Growth algorithm is performed. Note that while the FP Tree will contain items that are not part of the group (which are necessary because they occur as part of the patterns containing the items in the group), mining only needs to be performed for the actual group items. The output is the top-k list of patterns for all the items for each group (each group produces a top-k list for all items included in the patterns, even if they are not part of the group).

5. The resulting top-k lists from each group are aggregated to create the final result.

Figure 4.4 shows the MapReduce job structure of the Parallel FP Growth algorithm. The algorithm consists of four steps, using three MapReduce jobs:

Parallel counting This job scans the database, and creates the frequency list.

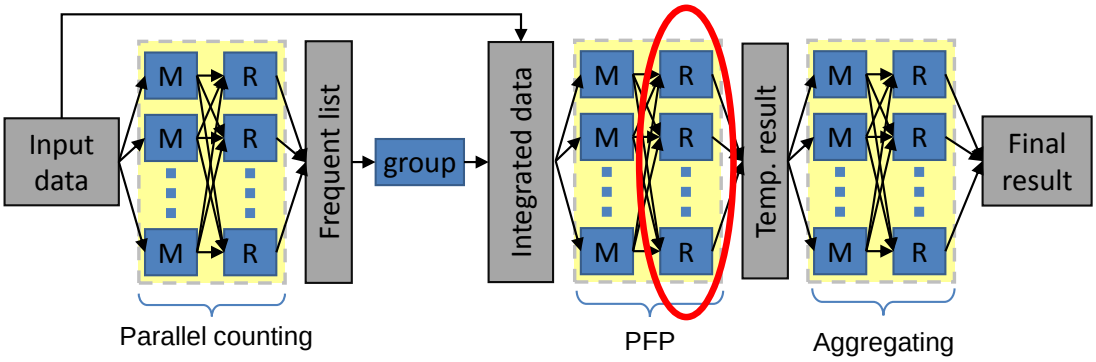


Figure 4.4: MapReduce job structure for Parallel FP Growth

Input data size	11 GB
Intermediate data size	120 GB
Output data size	10 MB
Map tasks	371
Reduce tasks	64

Table 4.1: Parallel FP Growth workload properties

Grouping Divide the frequency list into groups. This is done in a single worker and is not part of a MapReduce job.

PFP The main job of the algorithm; the map tasks generate group-dependent transactions, using the group ID as the key for the output. The reduce tasks perform FP Growth on the output.

Aggregating The temporary pattern lists are aggregated into the final result.

Much of the time of the overall algorithm is spent in the *PFP* job's reduce stage, which is highlighted in Figure 4.4. This stage is not only very lengthy, but depending on the input data and configuration it can be subject to fairly large data skew.

In order to demonstrate workload imbalance with this algorithm, I executed it using a synthetically generated transaction database created using the IBM Quest data generator [AS94]. The database contains 200,000,000 transactions with 100,000 unique items and an average transaction length of 10 items. In order to magnify the effect of the imbalance, I have altered the partitioning function for the algorithm.

The properties of the workload are given in Table 4.1. The workload was executed on a a 32 node cluster running Hadoop 0.20.203.0, using the hardware setup given in Table 4.2 for each node. Although an implementation of Parallel

CPU	2x Intel Xeon E5530 2.4GHz (8 cores)
Memory	24 GB
Disk	2x Seagate ST9500530NS (500 GB each)
Network	10 Gbps

Table 4.2: Experimental environment

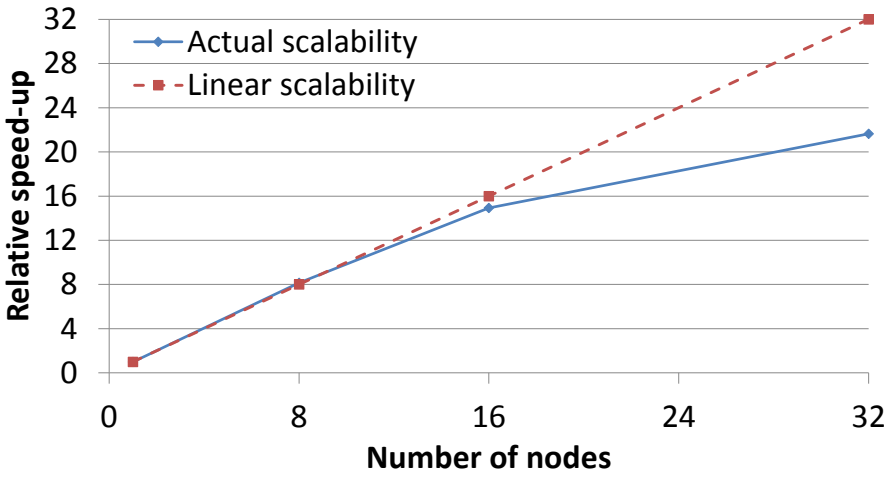


Figure 4.5: Scalability of Parallel FP Growth

FP Growth is provided in Mahout [Apac], due to issues with that implementation I have used a custom implementation instead.

Figure 4.5 shows how this workload scales when the number of nodes in the cluster is increased from 1 to 32. Up to 16 nodes, the workload scales relatively well because the number of reduce tasks is larger than the number of nodes (note however that due to the overhead per task, performance could be improved for lower numbers of nodes by using a lower number of tasks, provided that load imbalance would not be an issue). At 32 nodes, scalability drops of sharply, with the application only achieving a 21.6x speed-up compared to running on one node, rather than the desired 32x. Only a 1.4x speed-up is achieved going from 16 to 32 nodes, rather than the desired 2x.

This sub-linear scalability is almost entirely due to the data skew in the tasks. Figure 4.6 shows the distribution of the execution times of the tasks. There is a very large difference in the task execution times, with some tasks having almost no work to do and finishing in as little as 13 seconds, whereas the longest task takes 302 seconds, and it is that task that determines the execution time of the entire job. The dotted red line indicates the execution time of the job if the data was distributed equally amongst the task (this is estimated by using the average

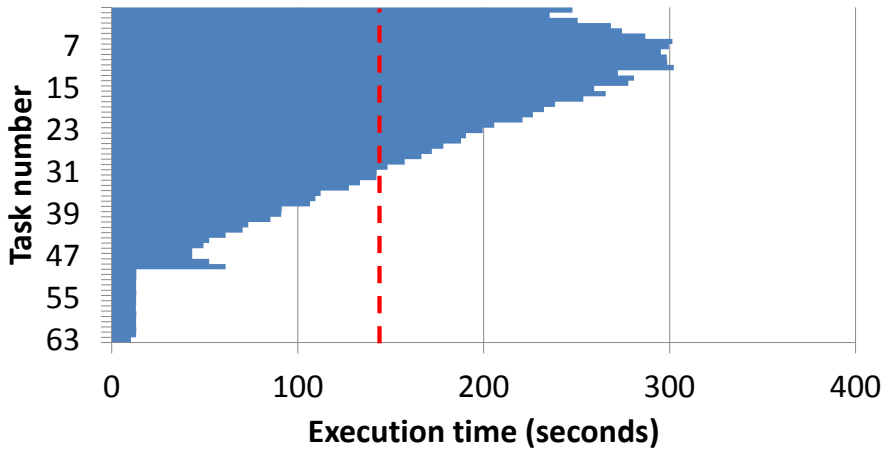


Figure 4.6: Task execution skew of Parallel FP Growth

execution time of the tasks), which is 145 seconds, a possible improvement of over 50%.

The effect of this imbalance on resource utilization can be seen in Figure 4.7, which shows the average CPU utilization across all nodes in the cluster for the entire Parallel FP Growth workload. The reduce tasks of the PFP job are highly CPU intensive, therefore this is an accurate indication of overall cluster utilization for that part of the workloads. It is clearly visible that during the PFP job's reduce stage, cluster utilization drops steadily as more and more tasks finish. This means that a lot of cluster nodes are idle, waiting for the final straggler tasks to finish.

4.4 Jumbo: an experimental platform for data intensive computing

In this section, I will introduce an experimental system for data intensive computation called Jumbo[1], that was created for the purpose of experimentation with workload imbalance issues and as a testbed for possible solutions.

The goals for the creation of Jumbo were as follows:

- To gain insight in the architecture, design, and development of a large scale data intensive distributed system, including the design decisions used by

[1] Jumbo is named after a 19th century male African Bush Elephant born in the French Sudan. Jumbo was an exceptionally large elephant made famous by the American showman P.T. Barnum, who brought him to America for exhibition. Jumbo's exceptional size gave rise to the word "jumbo", meaning large in size. See `http://en.wikipedia.org/wiki/Jumbo`. Hadoop was named after a toy elephant, hence the connection.

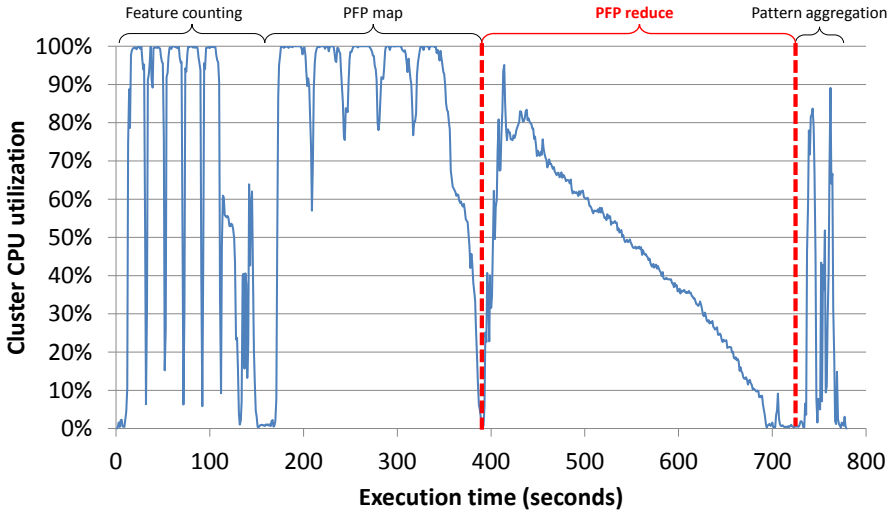


Figure 4.7: Average cluster CPU utilization of Parallel FP Growth

Hadoop.

- To perform experimentation for the purpose of determining the effects of workload imbalance and task skew on data intensive workloads.
- To provide a flexible testbed for possible solutions that is not hindered by Hadoop's rigid implementation of the MapReduce model.

Jumbo consists of a distributed file system and a programming model and execution environment for data intensive applications. Jumbo's design was heavily influenced by Hadoop because it was created in part to help understand the design of Hadoop. However, in some aspects it deviates considerably.

4.4.1 Jumbo DFS

The Jumbo Distributed File System is a DFS in the spirit of the Google File System (GFS) and the Hadoop Distributed File System (HDFS).

Jumbo DFS uses a single, centralized *NameServer* which maintains file system metadata, and which offers atomic operations for clients to manipulate the file system namespace (e.g. create or delete files and directories). File are divided into blocks, typically 64 MB or 128 MB in size, which are stored on *DataServers* in regular files, using replication to ensure fault tolerance. Blocks are further divided into packets of 64KB, which are checksummed to ensure data integrity. A packet is the smallest item of transfer in Jumbo: when reading or writing, you must always transfer an entire packet. Packet checksums are stored in packet headers inside

the block files to enable the file to be read sequentially (Hadoop stores checksums in a separate checksum file, requiring opening two regular files to read or write a DFS file).

Client operations

To write a file, a client communicates with the NameServer to create the file and its first block. The NameServer decides on replica placement for the block, and the client directly contacts the DataServer for the first replica in the list (which is ordered based on distance from the client) and sends it the file data. That DataServer forwards the data to the next replica.

When reading, a client requests the block list for a file from the NameServer, and then the replica list for the block it wishes to read. It then directly contacts the DataServer for one of those replicas to read data.

A client library abstracts these details so a client application can read and write the files as if they were normal local files.

Replica placement

Replica placement in Jumbo uses a policy that is similar to the policy used by Hadoop. If the writer is a node on the Jumbo cluster, the first replica is placed on the local server; if the writer is external, a random DataServer is selected. The second replica is placed on a different rack. The third replica is placed on the same rack as the first (note that when writing the replication list is ordered by distance so there is no unnecessary cross-rack communication). The remaining replicas (if any; the default number of replicas is 3) are placed at random. When selecting DataServers, the NameServer prefers those DataServers that have the lowest number of blocks currently open for writing. This helps to spread writing workload across the cluster.

Additional features

Jumbo DFS supports record aware streams. When writing a record aware stream, the client uses an API function to inform Jumbo where the record boundaries are. Currently, this information is used by Jumbo to ensure that a record does not cross a block boundary; if a record straddles a block boundary, it is moved to be entirely in the next block. This makes it much easier for clients that read only a single block (very common for MapReduce style clients) to determine where to start reading. This method can be extended to store record boundaries in packet headers to allow easy location of records at arbitrary file offsets, but this is not currently implemented. A file has to be explicitly marked to be record aware; by default, files do not use this option.

4.4.2 Jumbo Jet execution engine

Jumbo Jet[2] is the distributed execution engine for Jumbo. It supports the creation and execution of MapReduce style applications, but offers more flexibility to create more complex workloads, without the complexity of Dryad's directed acyclic graph job structure.

Job specification

A job in Jumbo consists of a linear pipeline of *stages*, connected by *channels*. Each stage reads input either from a channel or a file system like the Jumbo DFS (or possibly not at all, in the case of stages that just generate data based on some parameters), and writes data to a channel or a file system. A stage can have more than one channel as input to enable intrinsic support for joins. Stages are divided into *tasks* that each run a program on some piece of the input data. For stages that read from a file system, the input is split in some file system dependent manner (like Hadoop, Jumbo DFS uses blocks as the logical split boundary, optionally dividing each block into more than one split) and for stages that read from a channel, the data is partitioned by a user-specified partitioning function.

Tasks can perform arbitrary functions on the data. These can correspond to map or reduce functions, but other operators are also possible. This and the ability to chain together more than two stages gives Jumbo far more flexibility in representing complex algorithms that would have required a sequence of MapReduce jobs in Hadoop.

Channels are responsible for data transfer between stages, and for intermediate data handling. Jumbo supports three types of data transfer for channels (note: the stage writing to a channel is called the *sending stage* while the stage reading from the channel is the *receiving stage*):

File channels Data is stored in intermediate spill files on the local disk of the sending stage, and read remotely by the receiving stage. This functions in an identical fashion as the intermediate data shuffling functionality in Hadoop.

TCP channels Tasks in the sending and receiving stage establish a direct TCP connection and transfer buffered data over this connection. This requires that all the tasks in the receiving stage can be executed simultaneously, and also disables task-level fault tolerance.

In-process channels Two tasks are connected together to form a single compound task, and records are transferred via an in-memory buffer. Whereas

[2]Named after the nickname for the Boeing 747, an obvious connection once the base name Jumbo was decided.

the other channel types use an all-to-all connection between the tasks of the sending and receiving stage, in-process channels create a one-to-one connection between two tasks. The compound task is treated as a single task during execution.

Channels are also responsible for intermediate data handling, which controls any additional operations performed during transfer. By default, data is simply transferred unaltered and read by the receiving stage in the order the input segments were downloaded. This is useful for tasks that require no ordering guarantees at all, or that implement their own grouping. Jumbo includes built-in support for performing an external sort, in which case the output of the sending stage is sorted locally, and the input segments of a task in the receiving stage are merged before being passed to the task function. This is identical to the sorting operation performed on intermediate data by Hadoop.

Intermediate data handling is entirely customizable. While Jumbo only offers do nothing or external sort as built-in options, application developers can fully customize this to their application's needs.

Jumbo can easily mimic the behavior of Hadoop by utilizing a two stage job with a file channel which sorts the intermediate data and groups it by key on the receiving stage. However, it is able to utilize many alternatives that Hadoop's rigid implementation of the MapReduce framework does not allow. It can more naturally support joins and operators other than map or reduce, and it can represent complex algorithms in a single pipeline of stages rather than a sequence of separate jobs. In addition, jobs that do not require grouping or can utilize a more efficient method of grouping than sorting are free to do so, often enabling large gains in efficiency and also enabling tasks to perform more work with partial data, allowing tasks to better utilize resources even when all their input data is not yet available.

Job execution

Job execution in Jumbo Jet functions similar to Hadoop. A centralized *JobServer* listens for job submission and handles scheduling of tasks. *TaskServer* daemons run on each node to run tasks on behalf of the JobServer. A TaskServer checks each running task and reports to the JobServer if one crashes or stops progressing. If the JobServer receives such a report (or can no longer communicate with a TaskServer at all), it will automatically re-execute the appropriate tasks.

4.4.3 Programming model

Jumbo provides a programming model that allows for the specification of task programs and the structure of a job. One of the goals for Jumbo was that despite

the extra flexibility, programming a distributed processing application for Jumbo should not be considerably more difficult than writing one for Hadoop. The only piece of unavoidable complexity associated with Jumbo is knowing when to use that flexibility to customize your job, and when to stick to safe defaults. Jumbo provides some heuristics to automatically use e.g. improved grouping methods for common operations where they are known to be beneficial. This could likely be expanded on, but because this was not a significant target of this research this has not been explored further.

In this section, I will describe Jumbo's programming model with the intent that it should clarify the way Jumbo operates compared to Hadoop, and provide an insight into how the workloads used in later experiments are implemented on Jumbo.

Jumbo runs on Mono [Xam], an open source implementation of the Microsoft .Net Framework, and is written in C#. Writing distributed applications for Jumbo is therefore possible in any .Net language that is compatible with Mono, including C#, Visual Basic .Net, F#, IronPython, and others. All examples in this section use C#.

Tasks

The most important elements of any Jumbo job are the task programs that contain the core data processing functionality of the job. Unlike in MapReduce, Jumbo does not have distinct types of tasks; instead, all tasks are built using a common `ITask` interface, which is given below:

```
public interface ITask<TInput, TOutput>
{
  void Run(RecordReader<TInput> input, RecordWriter<TOutput> output);
}
```

A task program is simply a class that implements this interface. The `TInput` and `TOutput` generic parameters determine the input and output types (unlike in Hadoop, these don't have to be key/value pairs; any record type supported by Jumbo's lightweight binary serialization protocol is allowed). The `Run` method is provided with a `RecordReader` from which it reads the input, and a `RecordWriter` to which to write the output (these are abstract classes that will be instantiated with concrete types depending on the input and output).

For example, the class below implements the map class for the WordCount sample from Section 3.3.1:

```
class WordCountMapper
    : ITask<Utf8String, Pair<Utf8String, int>>
```

```
{
  public void Run(RecordReader<Utf8String> input,
                  RecordWriter<Pair<Utf8String, int>> output)
  {
    foreach( Utf8String record in input.EnumerateRecords() )
    {
      string[] words = record.ToString().Split(" ");
      foreach( string word in words )
      {
        output.WriteRecord(Pair.MakePair(new Utf8String(word), 1));
      }
    }
  }
}
```

For the sake of simplicity, certain common optimizations (such as reusing the object instance for the output record) were omitted from this sample.

At its base, every task program implements `ITask`. However, many more functionality can be optionally added. For example, a task class can implement `IConfigurable` to get access to the Jumbo configuration for the task, or `IHasAdditionalProgress` to report more detailed progress information. A task can utilize attributes to express for example that the input records can reuse the same object instance every time (something which Hadoop always does, but which due to Jumbo's looser semantics may not always be safe).

A number of abstract utility classes exist that implement `ITask` which other classes can derive from to gain some common functionality. Foremost among these is the `PushTask` class, which utilizes a push model for the input records rather than the pull model used by `ITask`. `PushTask` defines a method that processes a single record, much like a map function, for derived classes to implement. There are some places in Jumbo (such as tasks that receive data from an in-process channel) where tasks implementing `PushTask` offer some performance benefits over other types of tasks.

One of the utility classes is `ReduceTask`, which provides record grouping by key. It takes three generic parameters: the type of the input keys, the type of the input values, and the type of the output records. The following class implements the reduce function for the WordCount sample:

```
class WordCountReducer
    : ReduceTask<Utf8String, int, Pair<Utf8String, int>>
{
  public void Reduce(Utf8String key, IEnumerable<int> values,
                     RecordWriter<Pair<Utf8String, int> output)
```

```
  {
    output.WriteRecord(Pair.MakePair(key, values.Sum()));
  }
}
```

Job specification

A job in Jumbo is a sequence of stages. Specifying a job means specifying the configuration of each of the stages, including their input and output (which may be a channel to another stage or an file on the Jumbo DFS) and which class implements their task program. Also included is the configuration of the channels, which specifies things like the channel type, partitioning function, the number of partitions, and any special handling that needs to be performed on the intermediate data (like sorting).

Job configuration is specified using an XML file, and it is possible to write such a file by hand and submit the job that way, but this is not recommended. A second way to do it is to use the `JobConfiguration` class, which represents the configuration as an object structure with a one-to-one correspondence to the XML file. This class is used internally by Jumbo to read and write job configuration files, and can be used by the user to construct a job.

However, the recommended way to create a job configuration is to use the `JobBuilder` class. This class provides a procedural language that describes the job as a sequence of operations that will be translated into a job configuration. It is in some ways comparable to Pig Latin for Hadoop, except that it is done with method calls rather than a custom language. The `JobBuilder` class includes method calls for common operations that automatically create the appropriate structure for those operations. A single operation may involve multiple stages. For example, when using an aggregation operation the `JobBuilder` will create a stage connected via an in-process channel for local aggregation as well as another stage connected via a file channel for global aggregation.

The `JobBuilder` uses heuristics to determine things like the number of input splits or the number of partitions to use. If desired these can be overridden by the user.

For example, the following code defines the MapReduce version of WordCount in Jumbo, using the map and reduce classes defined earlier:

```
public void BuildJob(JobBuilder job)
{
  var input = job.Read("input.txt", typeof(LineRecordReader));
  var mapped = job.Process(input, typeof(WordCountMapper));
  var sorted = job.SpillSort(mapped, typeof(WordCountReducer));
  var reduced = job.Process(sorted, typeof(WordCountReducer));
```

```
  job.Write(reduced, "output", typeof(TextRecordWriter<>));
}
```

The `Process` function is used whenever you already have a task class for the operation you want to perform. The `SpillSort` function sorts the intermediate data using an external sort implementation similar to how Hadoop implements intermediate data sorting, optionally applying a combiner function. This automatically sets up the channel to sort the data on the map side, and merge it on the reduce side.

The end result of this sample is a version of WordCount that operates exactly like it would in Hadoop, and does not use any of the special features of Jumbo.

The `JobBuilder` contains two more major features to ease creation of jobs: it can accept methods, rather than classes, for the task programs, and it contains many utility methods for common types of operations, often with an accompanying custom method signature that the user should implement for this operation. Using these features, it is possible to define an alternative version of WordCount as follows (some concrete types have been omitted from this sample for readability):

```
public static void MapWords(Utf8String record,
                            RecordWriter<Pair<Utf8String, int>> output)
{
  output.WriteRecords(
    record.ToString()
          .Split(" ")
          .Select(w => Pair.MakePair(new Utf8String(w, 1)))
    );
}

public static int AggregateCounts(Utf8String key,
                                  int oldValue, int newValue)
{
  return oldValue + newValue;
}

public void BuildJob(JobBuilder job)
{
  var input = job.Read("input.txt", typeof(LineRecordReader));
  var mapped = job.Map(input, MapWords);
  var reduced = job.GroupAggregate(mapped, AggregateCounts);
  job.Write(reduced, "output", typeof(TextRecordWriter<>));
}
```

The `Map` function automatically generates a task class based on the map func-

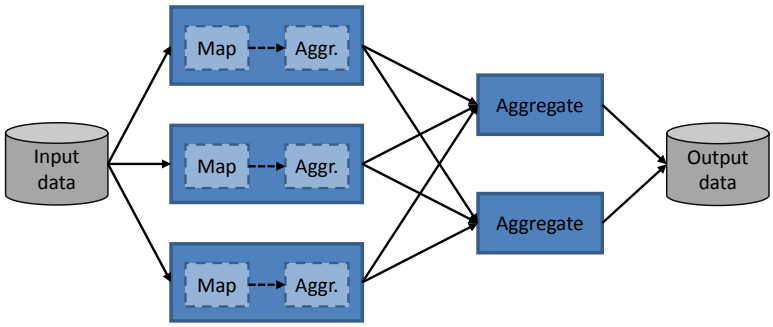


Figure 4.8: WordCount job structure in Jumbo

tion specified. This alleviates the user from the burden of writing the same plumbing for many tasks that can be trivially expressed in a single function.

The `GroupAggregate` function will aggregate the values for each key using a hash-table, which for WordCount is far more efficient than the sorting based grouping of regular MapReduce. It generates a task class to perform that hash-table based aggregation, requiring the user to only provide a function that performs the aggregation function given the old value stored in the hash-table and a new value from a record that was read from the input. In this case, that function is a simple sum of the values.

The job structure of this version of WordCount is shown in Figure 4.8. It consists of two main stages, the first of which uses a compound task consisting of a map task and an aggregation task connected using an in-process channel. That stage is connected via a file channel to a global aggregation stage. This figure uses three input splits and two partitions as an example; obviously those values will differ depending on the input data and system configuration.

Because the `JobBuilder` uses delegates to let the user specify which method to generate the class from, it is even possible to use lambda's in place of full-blown methods, further simplifying the job creation process.

Advanced features

Jumbo has many more ways to customize a job. A user can create classes derived from `RecordReader` and `RecordWriter` to implement custom input formats. You can also create special `RecordReader` classes that are responsible for combining input from multiple sources, either for a task that needs to merge the input segments of the previous stage or for a stage that receives inputs from multiple stages. Although a number of such classes are built in to support for example merge sort for intermediate data or inner-loop joins, this is fully customizable to allow any kind of data handling the user wishes for the job.

An in-depth explanation of these and other advanced features is beyond the scope of this thesis.

4.5 Dynamic Partition Assignment

As discussed in Section 4.3.2, the reduce stage of MapReduce jobs is particularly sensitive to data skew. Although it is possible to use sampling of the map output to adjust the partitions, this must be done before the job and can only help to mitigate data skew, not processing skew or performance heterogeneity.

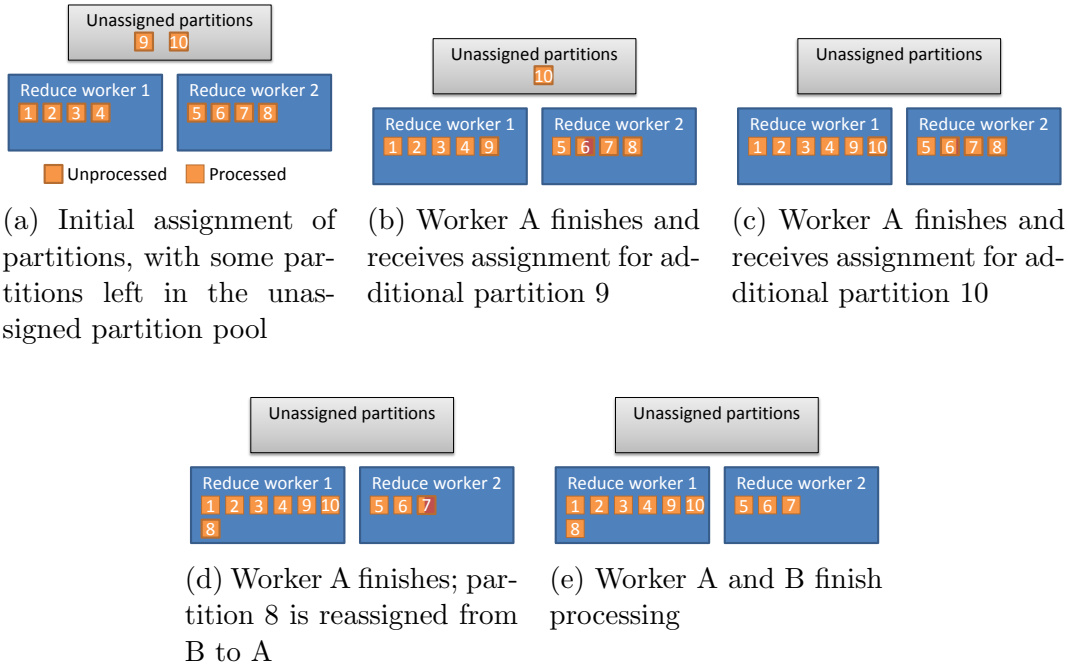
In Section 4.3.1, it was mentioned that one of the mitigating factors for stragglers in the map phase was that the large number of map tasks means that they are individually relatively short, reducing the relative impact stragglers have on the entire workload. While increasing the number of reduce tasks could have a similar effect, there are several reasons why this is not a desirable solution:

- Hadoop has a considerable overhead associated with the scheduling and execution of additional tasks. This includes overhead of starting the Java Virtual Machine, loading job configuration, and other start-up operations. For reduce tasks, the scheduling overhead is larger because Hadoop never schedules more than one reduce task for a Tasktracker on a single heartbeat. Additionally, the implementation of the shuffle phase also adds additional overhead that does not exist for map tasks. My experiments have shown that the overhead of increasing the number of tasks can be in excess of five seconds per task.

- More reduce tasks means they will be executed in more waves. Only the first wave can be started in the background while some map tasks of the same job may still be running, so adding additional waves reduces the amount of work that can be done on the reduce stage while waiting for map task stragglers.

- Every reduce task reads an input segment from every map task. The total number of transfers performed is therefore $M * R$ with M map tasks and R reduce tasks. Each of these transfers potentially causes a disk seek operation, and with increasing numbers of transfers the relative size of the data read for each disk seek becomes smaller. For large numbers of tasks, the effect of this can be significant. Hadoop also does not use persistent connections to the Tasktrackers for shuffling, adding network overhead to each transfer.

While the first issue is largely an implementation issue and indeed does not affect my Jumbo framework nearly as much as Hadoop, the other two issues are fundamental and cannot be circumvented without a radical change in shuffling architecture.



(a) Initial assignment of partitions, with some partitions left in the unassigned partition pool

(b) Worker A finishes and receives assignment for additional partition 9

(c) Worker A finishes and receives assignment for additional partition 10

(d) Worker A finishes; partition 8 is reassigned from B to A

(e) Worker A and B finish processing

Figure 4.9: Dynamic Partition Assignment in operation

What is needed therefore is a method to increase the granularity of workload distribution to achieve the same balancing effect of having more tasks, but without the added overhead of creating more tasks and performing more and smaller transfers. I propose a method for workload balancing called *Dynamic Partition Assignment* that is able to meet this goal.

The core principle of Dynamic Partition Assignment is to decouple the notions of partitions and tasks. In traditional MapReduce, each task processes exactly one partition. With Dynamic Partition Assignment, there are more partitions than tasks, so each task processes more than one partition and partitions can be dynamically reassigned to different tasks when imbalance is detected.

4.5.1 Dynamic Partition Assignment algorithm

Dynamic Partition Assignment works by having a pool of unassigned partitions from which tasks select partitions. Tasks are assigned partitions from the pool, or when the pool is depleted they can be assigned partitions from other tasks.

The algorithm uses the following steps, which are illustrated by Figure 4.9, which gives an example using a hypothetical job with 10 partitions and 2 reduce workers.

1. When the job is created, a user-specified percentage of the total number of

partitions is distributed evenly across the workers. In Figure 4.9a, this percentage was set at 80%, so each of the two workers is assigned four partitions to process, with two partitions remaining in the unassigned partition pool.

2. Each worker shuffles the data for all their assigned partitions. Data for sequentially numbered partitions is stored sequentially in the map task's spill files, so a single network transfer and disk seek can transfer all partitions in the initial assignment. A worker may also perform background merges on already shuffled partitions even if processing those partitions has not yet started.

3. Each worker starts processing their partitions one by one. Before a worker starts processing a partition, it checks that this partition is still assigned to it; in the case of a reassigned partition, it is skipped.

4. When a worker finishes processing all its assigned partitions, it requests assignment of additional partitions. If there are still partitions in the unassigned partition pool, it is assigned one of those partitions. This is illustrated in Figure 4.9b and c, where worker A finishes processing before worker B so it is assigned the two remaining partitions from the unassigned partition pool. The reason why worker A finished processing so much sooner than worker B is not important to the algorithm; it could be data skew, processing skew or hardware heterogeneity, but the handling is the same. Imbalance is detected simply by observing when some tasks finish sooner than others, and additional work is assigned to those tasks.

5. When a worker finishes processing and the unassigned partition pool is empty, a partition is reassigned from the worker with the most remaining unprocessed partitions. In Figure 4.9d, worker A exhausts the unassigned partition pool before worker B is finished, and since worker B has not yet started processing partition 8 it is reassigned to worker A.

6. When a worker finishes processing and no partitions can be reassigned, the worker finishes. When all workers finish, the job is finished. This is shown in Figure 4.9e.

The primary advantage of this approach compared to using more reduce tasks is that the transfer of the initially assigned partitions happens as a single sequential read and a single network transfer. This means that the average size of each read can be kept fairly large, minimizing the overhead of disk seeks. The larger the initial partition assignment percentage, the larger those initial reads will be and the fewer partitions have to be read one by one.

If a partition is reassigned from a task A to another task B, that partition is shuffled twice. Task A had already transferred all the data belonging to the partition, and may even already have performed background merges for that partition. However, when the partition is reassigned, task A will not process that partition so the data is discarded and any work done on shuffling and/or sorting that partition is not used. Task B will shuffle and sort the data for the partition on its own (the data is shuffled from the original sources of the data, not from task A, to avoid putting additional load on the node that is running task A in the case that task A's slowness was already caused by overloading of the node it was running on).

The initial partition assignment percentage therefore represents a trade-off between having larger initial transfers and having potentially more duplicate transfers. If you expect the tasks to be reasonably well balanced, a high percentage will enable efficient transfers with only a very small overhead compared to running in a default MapReduce configuration with a normal number of reduce tasks each processing a single partition.

Dynamic Partition Assignment can mitigate the effects of imbalance from data skew, processing skew or performance heterogeneity without any a priori knowledge about the hardware or data. The only requirement is that the data can in fact be split into smaller partitions. If there are any very large keys responsible for the skew, these cannot be split into smaller pieces so increasing the number of partitions does not offer any benefit in that scenario.

4.5.2 Implementation

I implemented Dynamic Partition Assignment on the Jumbo distributed processing system described in Section 4.4. I chose Jumbo as the test platform because its more flexible design mad it easier to implement Dynamic Partition Assignment, and it has reduced overhead allowing the benefit of Dynamic Partition Partition Assignment to be more clearly observed. Jumbo does not have an explicit notion of reduce tasks, so DPA can be applied to any channel regardless of whether the receiving stage performs a reduce or another type of operation.

During job construction, the user specifies the number of partitions and the number of receiving tasks for a channel. If the number of partitions is larger than the number of tasks, Dynamic Partition Assignment is enabled. The user also specifies the initial partition assignment percentage (if not specified for a job, the value from the system configuration is used).

The JobServer takes the role of partition manager. When a job is created, it creates a pool of partitions, takes the specified percentage of partitions from that pool and assigns them evenly among the tasks.

When a task starts running, it queries the JobServer for its assigned partitions and begins shuffling and sorting those partitions. Before actual processing of each

partition begins, it contacts the JobServer again to verify the partition has not been reassigned. If not, the task starts processing the partition. The JobServer also marks the partition as currently being processed so it cannot be reassigned (only partitions that are not being processed can be reassigned to another task). If a task is informed that a partition is reassigned, it simply discards all the data for that partition and skips to the next partition.

When a task finishes processing all its assigned partitions, it contacts the JobServer to request additional partition assignments. The JobServer assigns an additional partition from the unassigned pool or, if the pool is empty, searches the list of already assigned partitions for a partition to reassign. It will select the last partition of the task that has the most remaining unprocessed partitions. If multiple tasks have the same number of unprocessed partitions, one is selected at random.

Only a single new partition is assigned to a task at a time in the current implementation. When this partition is assigned to a task, it is immediately marked as being processed (even though the task still needs to shuffle it) to avoid partitions being reassigned back and forth between tasks.

If a task does not receive a new partition assignment, it finishes as usual.

In a normal MapReduce job, each task generates a single output file. In the case of a Jumbo task writing to the DFS, a file is created for each partition. If the task is writing to a channel, no distinction is made between the partitions as the channel re-partitions the records anyway.

Jumbo will create a new instance of the task class for each partition, so semantically it operates in the same way as having multiple tasks for each partition. A task class can use an attribute to indicate it's capable of handling all partitions in a single instance, allowing certain advanced optimizations such as sharing data structures between partitions.

4.5.3 Refinements

A number of refinements are possible for Dynamic Partition Assignment. These are not currently implemented and are left for future work.

- Use of output statistics: the JobServer could keep track of the size of each partition as each map task finishes. This information could be used to choose which partitions to reassign to minimize duplicated work.

- Bulk assignment of partitions: if a task finishes sufficiently ahead of other tasks, it may be preferable to assign multiple partitions to it at once, to retain the advantage of larger transfers for the dynamically assigned partitions. In combination with statistics gathered about partition size, this could be used

to combine multiple small partitions. Note that this is only beneficial for consecutively numbered partitions because otherwise they are not stored sequentially so cannot be transferred together without disk seeks.

- Ahead-of-time partition assignment: currently, a task can only request new partitions once it finishes. If additional processing capacity is available, it may be preferable to start shuffling and merging a new partition before the current one has fully finished processing. Determining when this would be beneficial and when it would cause additional interference would be a complex problem, however.

- Speculative execution of partitions: like the regular speculative execution of tasks, if a task is taking a long time to process a particular partition but there are no remaining partitions to reassign, other tasks could speculatively execute the same partition to see if they can process that partition faster.

- Guided initial partition assignment: in the case that information about data skew or hardware heterogeneity is available, the initial partition assignment could be altered to account for these known factors. For example, it could assign fewer initial partitions to tasks running on nodes with less powerful hardware specifications or utilize knowledge about relative partition sizes to assign partitions according to a bin packing algorithm.

4.6 Experimental evaluation

Dynamic Partition Assignment was evaluated using the Parallel FP Growth workload from Section 4.3.3. The experiment was performed on a 32 node cluster running Jumbo where each node has the hardware given in Table 4.2.

4.6.1 Workload

I used the Parallel FP Growth workload with as input data a randomly generated database [AS94] with 200,000,000 transactions, 100,000 unique items and an average of 10 items per transaction. The workload has the properties given in Table 4.1.

I implemented Parallel FP Growth workload for Jumbo, porting the custom implementation previously used for Hadoop in Section 4.3.3. The algorithm and implementation are identical, so any performance differences between Hadoop and Jumbo are purely due to efficiency improvements available in Jumbo.

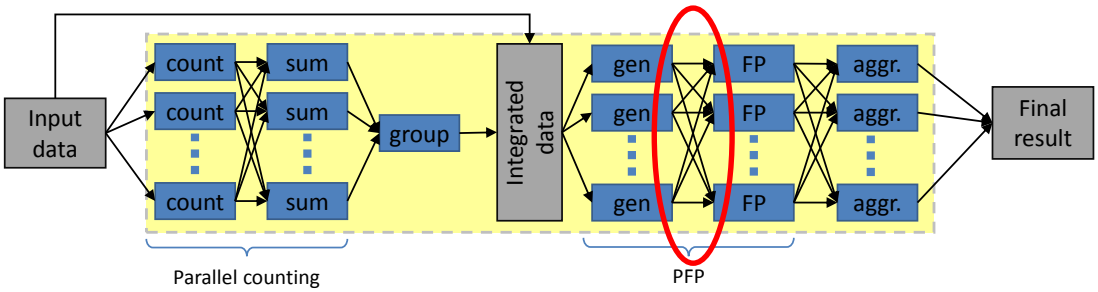


Figure 4.10: Parallel FP Growth job structure in Jumbo; DPA is applied to the highlighted channel

The structure of the Parallel FP Growth job in Jumbo is given in Figure 4.10; compare this to the Hadoop structure given in Figure 4.4. The Jumbo version uses a single job with six stages.

Count Read the transaction database from the DFS and uses a compound task that extracts items from the database and locally aggregates the number of occurrences of each item using hash-table based grouping. This allows for far more efficient grouping than the sorting-based grouping required by Hadoop. The output channel of this stage does not perform any special handling of the intermediate data (no sorting is used).

Sum Aggregates the global counts of all the item in the database using a hash-table. The output channel does not perform special handling of the intermediate data.

Group A single task (no parallelization) sorts the frequency list and divides it into the specified number of groups. The result is written to the DFS.

Gen Equivalent to the map stage of the *PFP* job in the MapReduce version. Reads the transaction database and grouped frequency list from the DFS, and generates group-dependent transactions. The number of partitions on the output channel is set to equal the number of groups, so each group-dependent transaction's partition is equal to their group number. This eliminates the need for any further grouping or sorting, so the output channel does not need to do any special handling of the intermediate data.

FP Equivalent to the reduce stage of the *PFP* job in the MapReduce version. Performs the FP Growth algorithm on each group. The output is a key/value pair with the item as the key and the item's top-k transactions as the value. The output channel does not do any special handling of the intermediate data, as grouping is handled by the next stage.

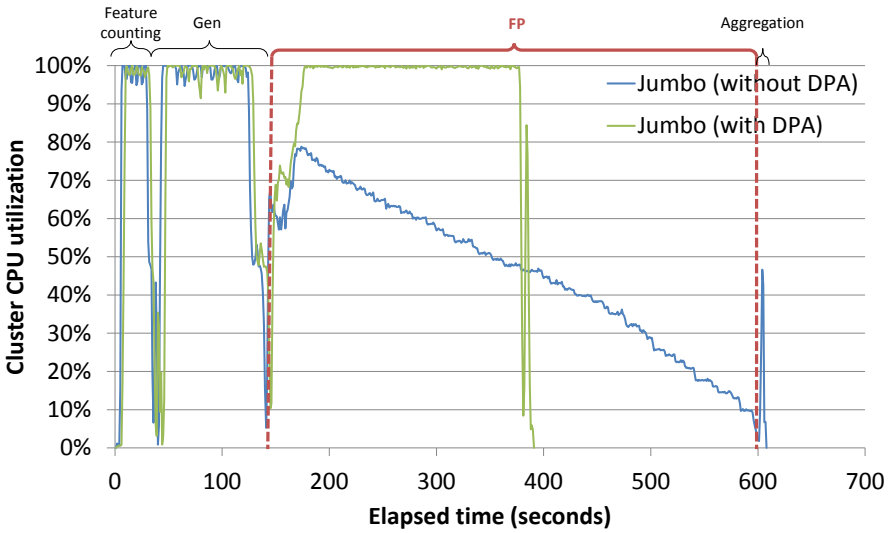


Figure 4.11: Average cluster CPU utilization of Parallel FP Growth with and without DPA

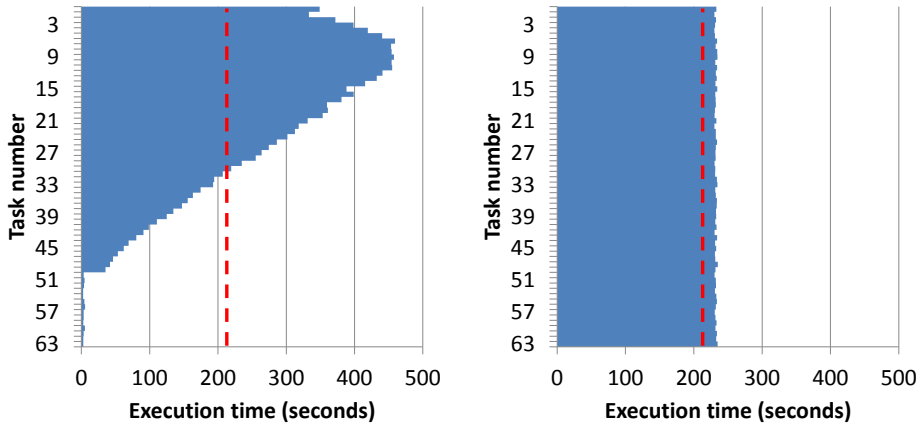
Aggregation Aggregates the frequent patterns for each item. The output is written to the DFS.

Note that for each of the stages, it was possible to use a more efficient grouping method other than sorting. No sorting of the intermediate data is done at any point in the job, leading to a considerable improvement of efficiency over the plain MapReduce version. Additionally, the map stage of the MapReduce aggregation job was removed entirely, because it served only to repartition the temporary result. Because the FP stage writes directly to a channel and not the DFS, it is already capable of doing this repartitioning without needing an extra stage. These changes reduce the overall execution time of the job, and therefore increase the relative effect of stragglers making the benefit of Dynamic Partition Assignment more clear.

The *FP* stage is the equivalent of the reduce stage of the *PFP* job in the MapReduce version of the algorithm, and since I use the same input data and job configuration it is subject to the same amounts of data skew. Dynamic Partition Assignment is enabled on the channel between the *Gen* and *FP* stages to mitigate the skew.

4.6.2 Results

I compared the result of running the workload using 32 nodes with Dynamic Partition Assignment enabled and disabled. I used 7680 groups and therefore also



(a) DPA disabled. (b) DPA enabled.

Figure 4.12: Execution time of each task of the *FP* stage with DPA enabled and disabled, where the red line shows the optimal execution time

7680 partitions, and 90% initial partition assignment. This is compared against an execution where Dynamic Partition Assignment is disabled: in this case, initial partition assignment is 100% and reassignment is not allowed.

Figure 4.11 shows the average cluster CPU utilization across all nodes. As before, since the *FP* stage is largely CPU bound (particularly during the actual processing of the FP Growth algorithm) this is an accurate measure of overall cluster utilization during this part of the job. The line where DPA is disabled shows a similar profile to the MapReduce result of Figure 4.7, with cluster utilization dropping as more tasks finish processing. Because the improved efficiency of Jumbo has drastically reduced the execution time for the feature counting, generating group-dependent transactions, and aggregation steps, the relative length of the *FP* stage is longer and the relative impact of the under-utilization appears larger.

With Dynamic Partition Assignment enabled, cluster utilization remains at 100% for nearly the entire *FP* stage, and all tasks finish nearly simultaneously (the final CPU spike is the aggregation stage). The execution time of the *FP* stage was reduced by 50% from 460 seconds to 230 seconds, and the execution time of the overall workload was reduced by 36% from 608 seconds to 391 seconds. Additionally, Jumbo is 22% faster than Hadoop when DPA is not used, and 50% faster than Hadoop when DPA is enabled (note that this difference is somewhat smaller than it could be because the FP Growth algorithm itself runs somewhat slower under Mono than the equivalent Java version).

Figure 4.12 shows the individual execution time of each task with Dynamic Partition Assignment disabled and enabled. When Dynamic Partition Assignment is not used, Jumbo shows a similar distribution of task times as Hadoop in

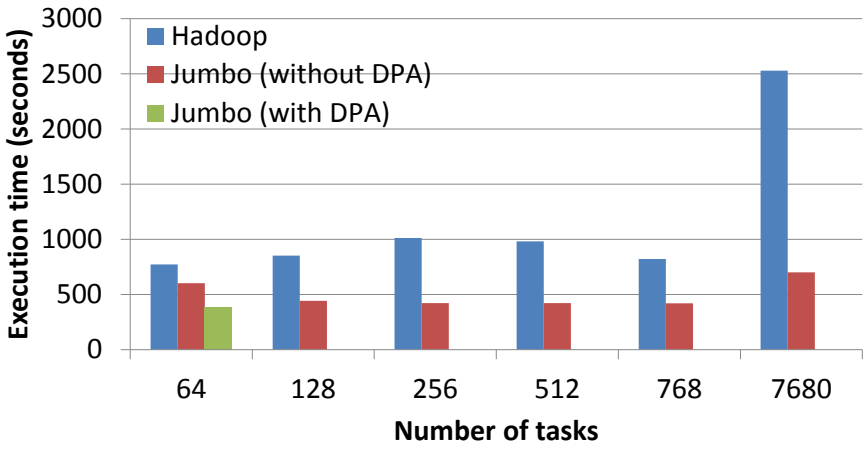


Figure 4.13: Comparison of Dynamic Partition Assignment versus increasing the number of tasks

Figure 4.6. The dotted red line indicates the optimal execution time, derived by taking the average execution time of each task in Figure 4.12a.

It can be seen from Figure 4.12b that with Dynamic Partition Assignment enabled, the actual execution time of 230 seconds closely approximates the optimal execution time of 212 seconds. The difference is due to the overhead of duplicate transfers, individually transferring the dynamically assigned partitions, and some minor implementation overhead for Dynamic Partition Assignment.

It is clear that Dynamic Partition Assignment can greatly reduce the effects of data skew on a workload, with only minimal overhead. The execution time of the affected stage in this example was halved by the use of DPA, significantly improving the overall execution time of the workload. The cluster was fully utilized for almost the entire workload, which effectively means that very few resources were wasted, enabling a nearly optimal utilization; further improvements are only really possible by increasing the workload's efficiency, not by improving its resource utilization.

DPA compared to increased task numbers

The alternative option for increasing load balancing granularity is to increase the number of reduce tasks. To test how Dynamic Partition Assignment compares to this approach, I executed the Parallel FP Growth workload using 128, 256, 512, 768, and 7680 tasks in the FP stage (Jumbo) or PFP reduce stage (Hadoop). Figure 4.13 shows the execution time of the entire Parallel FP Growth workload (including all jobs and stages) using these different settings, comparing the result to the original 64 task workload with imbalance and when using Dynamic Partition

Assignment.

The results of using 128, 256, 512 or 768 tasks for Jumbo are all very close together, and also quite close to using Dynamic Partition Assignment; they are between 10 and 20% slower than using DPA, with 128 tasks being the slowest and 768 tasks the fastest (note that these percentages relate to the entire workload, not just the affected stage). It appears that increasing the number of tasks in this case is mostly successful in reducing the imbalance and eliminating stragglers. With 128 tasks there is still some imbalance, and increasing to 768 tasks the balance improvements still outweigh the reduced efficiency due to per-task overhead. Note that 256, 512 and 768 tasks are within 1% of each other, so it unlikely that much more improvement can be gained form further increases.

For Hadoop, the overhead of increasing the number of tasks is large enough that even just using 128 tasks is 10% slower than the original imbalanced version with 64 tasks. For 256 and 512 tasks, the per-task overhead dominates, further slowing the job down. At 768 tasks Hadoop seems to hit a sweet spot between improved balance and task overhead, improving performance somewhat, although even that setting is still 6% slower than using 64 tasks. Part of the overhead is also caused by the fact that, due to the multi-job structure of Parallel FP Growth in Hadoop, increasing the number of reduce tasks for the second job also increases the number of map tasks in the third job. Despite seemingly reducing imbalance compared to the original case, increasing the number of tasks actually reduces the efficiency of the job.

Since I used 7680 partitions for Dynamic Partitions Assignment, to really get the same level of load balancing granularity that Dynamic Partition Assignment provides it is necessary to also use 7680 tasks. In this case, the overhead per task means that both Jumbo and Hadoop are slower than their original imbalanced executions with 64 tasks. Hadoop is particularly bad in this case with a 227% increase in execution time compared to 16% for Jumbo. Jumbo with 7680 tasks is 83% slower than using Dynamic Partition Assignment.

The reason that for this workload, at least for Jumbo, a small increase in the number of tasks appears quite effective in reducing imbalance is largely due to the structure of the workload. Both Jumbo and Hadoop schedule reduce tasks in numerical order, and Figure 4.12a shows that the tasks in this workload are roughly ordered from longest to shortest which makes it much easier to balance when the number of tasks is increased. This is indeed observed during execution; the nodes that run the shorter tasks from the first wave are able to complete most or all of the really short tasks before the really long ones from the first wave finish.

In general, it is not always possible to order tasks this way. That it happens here is a coincidence, a natural consequence of the relation between the item frequency in a group and the length of the transactions in a group (while higher-numbered groups have longer transactions due to the way group-dependent transactions are

generated, they have far fewer of them). For other workloads that may not have this property by themselves, ordering the tasks this way would require knowing the relative sizes of the partitions. This can only be determined by sampling the intermediate data using an additional execution of the job. Even then, this only addresses the problem for data skew, not processing skew or hardware heterogeneity.

Even for a workload like this, where tasks are ordered such that only a small increase in the number of tasks can gain large improvements in balance, there is still a key advantage to using Dynamic Partition Assignment: it can be enabled even if the user doesn't know if it will be necessary, without worrying about too much additional overhead. The number of reduce tasks is something that must be configured by the user or system administrator, and the rule of thumb says you should have at most one or two waves. Especially for Hadoop, where the overhead of increasing the number of tasks is much larger than for Jumbo, this rule is important. Unless the user already knows that the workload is going to be imbalanced, there is no reason to increase the number of tasks because the expectation is that this decreases performance. Even if the user does know, it would require careful experimentation to determine the optimal number of tasks. By contrast, with Dynamic Partition Assignment a very large number of partitions can be used without significant overhead. This means the user can enable Dynamic Partition Assignment knowing that if there is imbalance it can be reduced, but that if there is no imbalance, no harm will be done. It removes the need for the user to decide whether the trade-off between overhead and potential balance improvements is worth it.

4.6.3 Performance heterogeneity

The previous section demonstrates how Dynamic Partition Assignment can mitigate the effects of data skew and processing skew. I also evaluated its ability to address the effects of performance heterogeneity.

For this purpose, I used a 48 node cluster using the node types described in Table 4.3, with 32 nodes of type A and 16 nodes of type B. The workload used a transaction database of 800,000,000 items with the properties shown in Table 4.4, and was configured to *not* have the same kind of data skew as the previous experiment. Every node runs at most two simultaneous tasks (including the type B nodes) to avoid excess I/O interference. This means that of the 192 reduce tasks, 64 were executed on the type B nodes and the remaining 128 on the type A nodes.

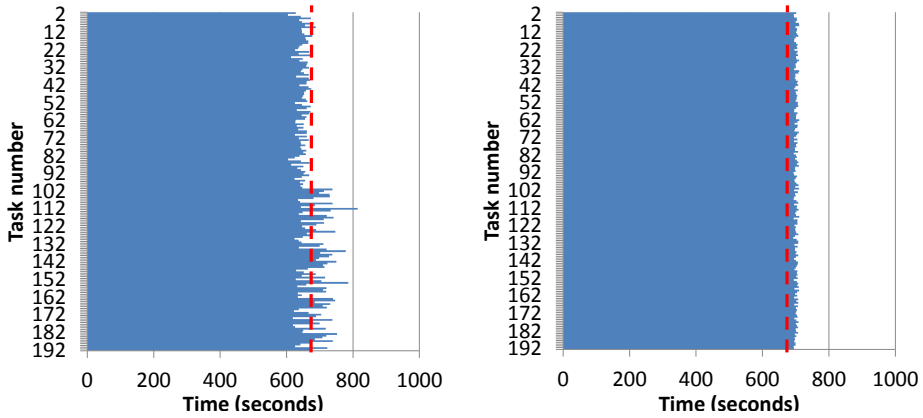
The execution times of each task of this workload with DPA enabled and disabled are shown in Figure 4.14. The variance in the task times is far less in Figure 4.14a than in Figure 4.12a because there is no data skew and the practical

Node type A	
CPU	1x Intel Core 2 Duo E6400 2.13GHz (2 cores)
Memory	4 GB
Disk	1x 1 TB
Node type B	
CPU	2x Intel Xeon E5410 2.33GHz (8 cores)
Memory	32 GB
Disk	2x 1 TB

Table 4.3: Experimental environment for performance heterogeneity

Input data size	44 GB
Intermediate data size	240 GB
Output data size	10 MB
Map tasks	1484
Reduce tasks	192

Table 4.4: Parallel FP Growth workload properties for performance heterogeneity



(a) DPA disabled. (b) DPA enabled.

Figure 4.14: Execution time on the heterogeneous environment of each task of the *FP* stage with DPA enabled and disabled, where the red line shows the optimal execution time

difference in performance between the nodes is not that big for this workload. The *FP* stage is mostly CPU bound and since each task effectively uses one CPU core, the most relevant metric here is the difference in speed between each core of node types A and B, which is not that big. Nevertheless, there are clear stragglers holding up the entire workload, with a total execution time for the *FP* stage of 814 seconds, where optimally it would have been 668 seconds if all tasks had equal execution times.

Figure 4.14b shows that enabling Dynamic Partition Assignment is able to reduce the variance in task times, and improve the execution time of the *FP* stage by 14% to 707 seconds, which is within 6% of the optimal execution time.

This demonstrates that Dynamic Partition Assignment is also able to mitigate the effects of performance heterogeneity, something which sampling-based approaches to reducing data skew are not able to do. The benefit of Dynamic Partition Assignment would be bigger if the performance difference between the nodes was bigger, but unfortunately this was the only heterogeneous environment available to me at the time of this experiment.

4.7 Conclusion

In this chapter, I have shown how data skew, processing skew and performance heterogeneity can cause stragglers in MapReduce, and how this can hinder resource utilization on large clusters. The design of MapReduce means that a straggler in the map stage will prevent the reduce stage from progressing, and a straggler in the reduce stage will delay completion of the workload, and prevent subsequent jobs that depend on the output of that job from starting. Stragglers in the reduce stage are particularly common due to the low number and relatively long length of tasks, and the reliance on a user-specified partitioning function to divide work amongst tasks.

While a job is stuck waiting for stragglers, many of the resources of the cluster are not fully utilized and do not contribute to the progress of the job. Stragglers therefore prevent a workload from scaling linearly when additional resources are added to the cluster, because not all of the new resources can be fully utilized for the entire duration of the workload. In the cloud, where billing it typically based on the amount of time a resource is held by a user, this translates to additional costs for no extra benefit. The resources, although not fully utilized, can typically not be released because they can still be partially utilized or contain data that the rest of the workload requires to progress.

While running multiple workloads—possibly from multiple users—on one MapReduce cluster may help to utilize the spare resources, stragglers can still pose a problem if performance guarantees or QoS contracts need to be met for certain

jobs.

I proposed Dynamic Partition Assignment, a method that is able to alleviate the problem of stragglers in the reduce phase by increasing the number of partitions, and thus the load balancing granularity. By decoupling partitions from tasks and using a combination of large initial partition assignments that can be transferred using a single sequential read, additional partition assignments from a pool of unassigned partitions, and partition reassignment from stragglers to already finished tasks, Dynamic Partition Assignment is able to achieve this extra granularity without the overhead of increasing the number of reduce tasks.

Dynamic Partition Assignment was implemented on Jumbo, my experimental platform for data intensive computing, and tested using Parallel FP Growth as an example workload, both by introducing deliberate data skew and by testing using a heterogeneous cluster environment. In both cases, Dynamic Partition Assignment was able to reduce the variance in the task execution times and eliminate stragglers, improving performance by as much as 40% and bringing the affected stage's execution time to within 10% of the optimal execution time.

Unlike approaches that rely on sampling or gathering of statistics, Dynamic Partition Assignment is a no-knowledge load balancing approach that can respond to unexpected causes of task skew, including sudden performance degradation of some nodes, for instance due to background tasks or interference from other virtual machines sharing the same physical host.

Although Dynamic Partition Assignment was demonstrated on a MapReduce style system and workload, the issue of stragglers is a common one in many cloud applications. In any such application that relies on data partitioning, Dynamic Partition Assignment can be utilized to reduce the number of stragglers. The only requirement is that the data can be split into sufficiently small partitions to achieve the necessary granularity in balancing the workload. Extending this approach to handle very large single keys is left for future work.

Chapter 5

I/O interference in data intensive distributed processing

5.1 Introduction

In Chapter 4, I covered three major factors that can lead to inefficient resource utilization: data skew, processing skew, and performance heterogeneity. There is however another, altogether more complex, factor that can have a major impact on performance and resource utilization. This factor is interference between processes.

Interference occurs whenever two or more processes are attempting to access the same limited resource. Such a resource may be a local resource under contention from multiple processes on the same system, or a global resource under contention from processes running on different systems in the cluster. Processes may be tasks from the same workload, different workloads running in the same instances, or there can even be interference between virtual machines running on the same physical host.

Modern server nodes typically have many CPU cores, so it is desirable to use increased levels of parallelism on each node to be able to utilize the full CPU power of the node. However, data intensive applications such as those running in MapReduce or similar frameworks are often dominated by I/O operations, particularly reading and writing data from disks, and these I/O resources are usually more limited than the CPU resources. Disk storage solutions in the cloud can come in many forms: among other options it can be a single, traditional hard disk with spinning platters, it can be a high-performance SSD drive, or it could be a large storage array (e.g. RAID) attached to a node via a NAS or SAN. However, regardless of which is the case, they are often presented to the node as a single logical device that all processes must share.

On a system with multiple CPU cores, parallel processes (as long as the number

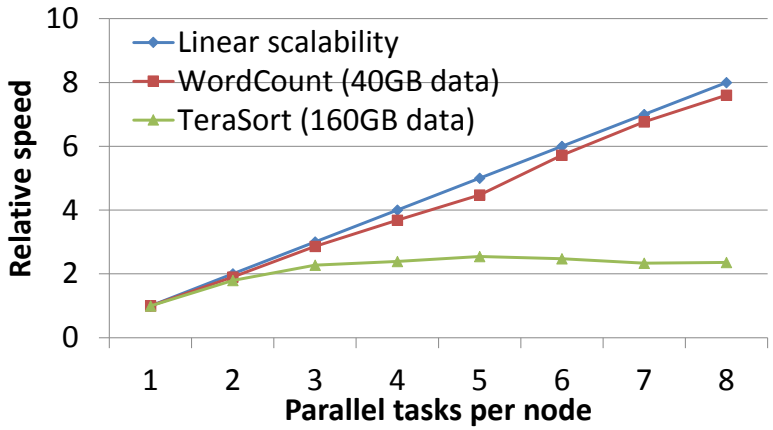


Figure 5.1: Scalability of WordCount and TeraSort with per-node parallelism

of threads does not exceed the number of cores) can be thought to each operate on a different CPU. Even though in reality they may be sharing some resources and shifting to different cores on occasion depending on scheduler behavior, there is only a small amount of interference at the CPU level because a dedicated set of CPU resources is available to each process at any given time. However, with only a single storage device I/O requests must be multiplexed between the processes. When multiple processes access such a device, this can lead to significant performance degradation. At best, the aggregate performance of all the parallel processes will be the same as when a single process accessed the device, but due to the properties of hard disks performance can often degrade even further.

Figure 5.1 shows an example of the scalability of two typical MapReduce workloads running on Hadoop 0.20.203.0 when the number of parallel tasks running on each node is increased from 1 to 8. The experiment was run on a cluster with nodes that have 8 CPU cores and 1 storage device (see Table 5.4), so when increasing the number of tasks to 8, performance reduction due to CPU contention should be minimal.

The first workload is WordCount, a canonical MapReduce workload taken from the original MapReduce paper [DG04] and provided in Hadoop [Apaa] as sample code. WordCount must parse the input to find the word boundaries, which uses a lot of string operations. String operations in Java are known to be expensive, particularly in a case such as this where a large number of temporary String objects are created, putting pressure on the garbage collector. Additionally, WordCount produces a very large number of intermediate records that must be serialized and lexicographically sorted. It also utilizes a combiner function in the map tasks to perform local aggregation of the counts for various words, meaning that both the intermediate data and output data of the job is very small. As a result, this

workload is almost entirely CPU bound. This is reflected in Figure 5.1, which shows that WordCount is able to scale almost linearly when per-node parallelism is increased. When running 8 tasks per node, WordCount runs 7.6x faster than when running a single task per node, 95% efficiency compared to linear scalability, indicating it is able to fully use the system's 8 cores.

The second workload is a sorting operation using the data format used by the TeraSort benchmark [Ter]. This format used fixed-size records of 100 bytes with a 10 byte key, meaning that they can be efficiently parsed and sorted quite easily. The map and reduce functions for this operation are no-ops, and the intermediate data and output data is the same size as the input data (no compression is employed). This workload is therefore very I/O intensive, which is clearly visible in Figure 5.1. It is only able to run 2.4x faster with 8 tasks per node, an efficiency of only 29% compared to linear scalability. Furthermore, it can be observed that peak performance was obtained running 5 nodes per task; when the number of tasks per node is increased further, performance actually decreases.

It is very difficult to successfully mitigate I/O interference with an approach like Dynamic Partition Assignment proposed in Chapter 4. For example, in the Parallel FP Growth experiments I found that if a large number of tasks per node was used sometimes the Hadoop or Jumbo scheduler would put the largest of the *PFP* reduce tasks on the same node, causing very large I/O interference on that node. When other tasks finish, Dynamic Partition Assignment would take away work from the tasks experiencing interference, but this only means that they will finish sooner, it does not alleviate the load of the system while those tasks are still processing, so interference will continue to occur until those tasks run out of work. While Dynamic Partition Assignment was able to reduce the variance between the tasks and avoid stragglers, the nodes experiencing interference still were not able to utilize all CPU resources and ran at less than optimally efficient processing speeds for the entire duration of those tasks, even though there were no stragglers.

Additionally, dynamically assigning partitions has the added effect of causing additional I/O, particularly in the case when reassigning a partition from one task to another, in some cases exacerbating the situation. Even if it was possible to instantaneously reduce the workload of the nodes experiencing interference, moving work away from those nodes would cause additional I/O

These examples demonstrate how I/O interference can prevent a system from being fully utilized. As data intensive applications are more and more pervasive among cloud workloads, it is increasingly important to be able to address the performance problems. My experience with Dynamic Partition Assignment shows it is very complex to mitigate I/O interference at the point it is detected, since moving work may cause additional I/O. It is therefore clearly necessary to be able to predict the effects of I/O interference before they happen.

Predicting I/O interference is a very complex problem. The behavior of hard

disks in particular is very complex due to their mechanical nature, but there are a number of other factors that affect the performance under interference:

- A large number of components are involved in determining how concurrent I/O operations interact, including the kernel I/O scheduler, read-ahead buffers, and any further scheduling or buffering done by the underlying hardware. In addition, the kernel CPU scheduler and the I/O scheduler can have complex interactions.
- A process's I/O usage may not be uniform over its entire execution. When two processes overlap, even if those processes perform identical or similar work, they may not be performing the same I/O operations at the same time.
- Environmental factors such as file system fragmentation can affect performance and I/O interference. Not only does reading a fragmented file affect performance, but concurrently writing multiple files often causes additional fragmentation on many common file systems.

In this chapter, I analyze the I/O behavior of MapReduce in detail and propose an I/O cost model that combines the application behavior with hardware models derived from micro-benchmarks to predict the effect of I/O interference. This model is evaluated with several representative workloads on a real cluster, and the results show it is able to make predictions with very high accuracy.

5.2 Related Work

The difficulty of efficiently using MapReduce has been recognized for some time. There have been a number of works that attempt to automatically optimize MapReduce execution [KPP09, WLMO11, JCR11, WOSB11, HDB11, JQRD11], typically focusing on optimizing query execution and sharing work between workloads, or automating cluster provisioning and Hadoop configuration. Other works focus on optimizing the I/O efficiency of MapReduce [RLC+12, RRS+12]. In these works, I/O is only considered when the goal is to reduce the total amount of I/O performed by the application or optimize the I/O patterns employed by MapReduce; the impact of I/O interference on existing MapReduce applications is not considered.

5.2.1 MapReduce modeling

Recently, there has been an increasing amount of work in modeling the behavior of MapReduce. Huai et al. [HLZ+11] propose a model that generalizes MapReduce

and similar frameworks into a matrix-based representation of the data flow. Verma et al. [VCC11a] propose a model to estimate job completion time based on the observed task completion times measured previously. Jindal et al. [JQRD11] model the read I/O behavior of map tasks. Yang et al. [YLLQ12] propose a statistical model to evaluate the effect of various configuration parameters.

The model proposed by Herodotou et al. [HDB11, Her10] is to my knowledge the most complete analytical model of Hadoop, focusing on relative performance between environments and optimizing Hadoop configuration. The data flow model for MapReduce execution used in this work is very comprehensive, and my model builds on this to some extent.

My work focuses on the I/O performance of MapReduce and differs from these existing models because all of them make simplifying assumptions about I/O or ignore it entirely.

5.2.2 I/O interference

For I/O interference itself, there has been a considerable effort to characterize and predict I/O performance [PLM+10, MWS+07], which focuses primarily on the hardware environment.

Chiang et al. [CH11] propose a scheduling system that takes I/O interference into account, but unlike my work it applies to VM scheduling rather than application scheduling, and was only evaluated using simulation.

I/O interference has also been considered in the context of automated storage management. BASIL [GKAK10] and Pesto [GSA+11] utilize models of the storage system that can be used to predict latency and load balance placement of virtual hard disks in a storage system. Also focused on virtual machine migration is Romano [PAL12], which constructs workload and hardware models for latency prediction on the fly by monitoring parameters such as I/O request size and queue length, and also includes an aggregation model specifically intended for dealing with I/O interference. Because these models focus on latency rather than application performance prediction, they are not directly comparable to my approach.

Related to my work is the work of Shan et al. [SAS08], which characterizes the entire I/O behavior of an application using a micro-benchmark; this differs from my work because I use micro-benchmarks only to establish basic hardware I/O performance and use an analytical model to describe the application's behavior. They also target more traditional supercomputers rather than cloud environments.

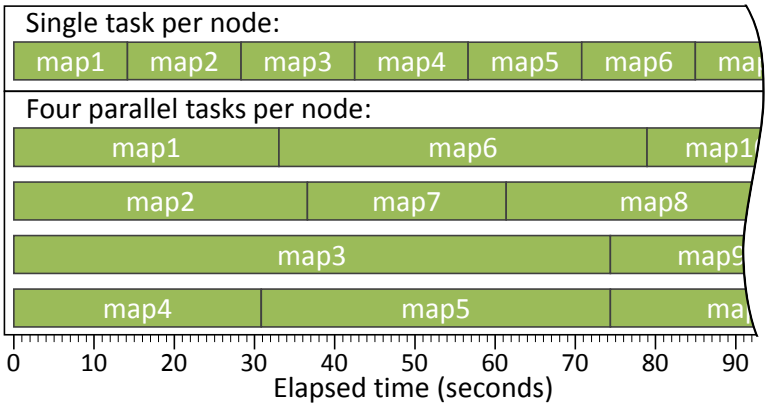


Figure 5.2: Partial time-line of map task execution of the TeraSort workload

5.3 Effects of I/O interference

A MapReduce application consists of many map and reduce tasks that read and write data on the Distributed File System (DFS) and local disks. In order to utilize multi-core systems, Hadoop schedules multiple tasks to run simultaneously on each node. In this case, the node's I/O resources are divided between these tasks.

Map and reduce tasks perform a combination of CPU and I/O processing. Because MapReduce jobs often process very large amounts of data, I/O operations often dominate the workload and their performance has a significant impact on the overall workload. Even if the map and reduce function used by the workload are very CPU intensive, some parts of the workload that are not directly involved with the execution of those functions are likely to be I/O bound as long as the data is big enough.

Figure 5.2 shows a partial time-line for one of the nodes in a cluster running the TeraSort workload. This time-line was taken from an actual execution of the workload running on the cluster. When the number of tasks that a node executes in parallel is increased from only a single task to four parallel tasks, the execution time of each task increases. In addition, a high degree of variation is observed in the tasks, even though when running only a single task at a time they were all nearly the same duration. Although this figure only shows what happens to map tasks on a single node, the same effect was observed for map tasks on all of the nodes, and was also observed to occur with reduce tasks.

The cause of the increased execution time can be seen when looking at the CPU and I/O cost for the tasks. The *CPU cost* is the time it takes for a task's CPU processing to complete, including any time it spends waiting to be scheduled on a CPU. Any time the task is not performing or waiting for a CPU operation

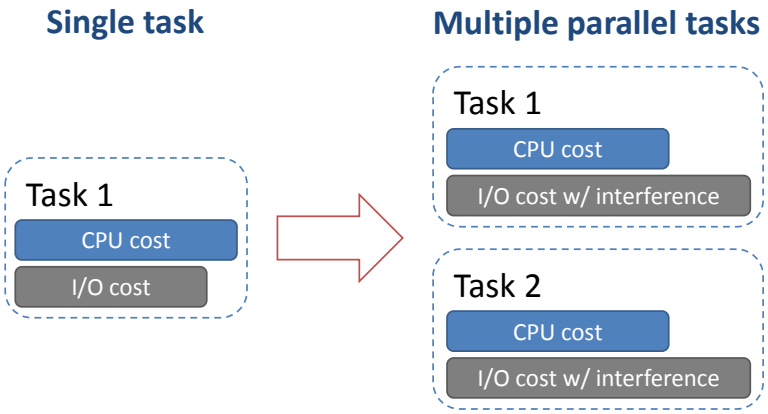


Figure 5.3: Per-task cost increase when I/O interference occurs

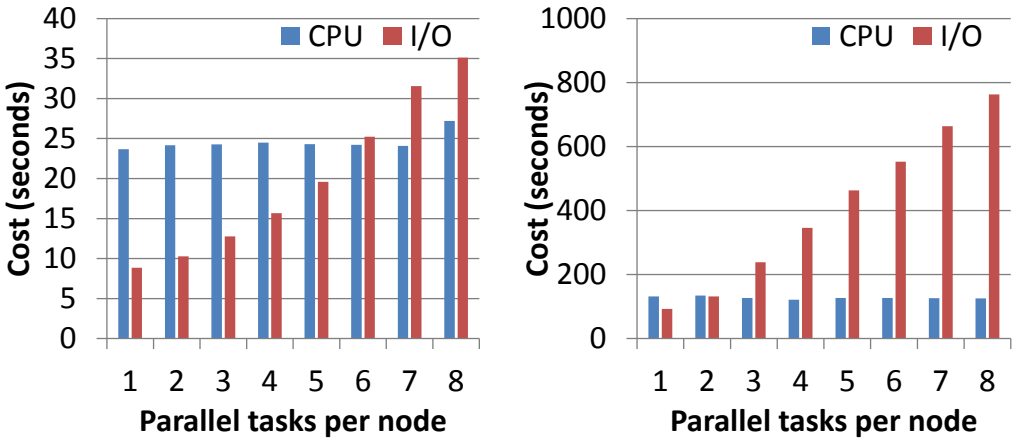
(for example when it is waiting for an I/O request to complete) is not included in these costs. Basically, CPU cost is the time that any CPU is busy performing operations for the task plus the time the task spends waiting for a CPU.

The *I/O cost* is the time it takes for I/O devices to finish servicing requests from the tasks. This may include time that an I/O request is queued up but not yet being executed. As with CPU cost, the I/O cost can be basically said to be the time that any I/O device is busy servicing request for the tasks plus the time that I/O requests are waiting to be serviced. I only consider disk I/O here, other types of I/O like memory or network are not included.

Since the CPU cost and I/O cost include the time spent waiting for devices to become available, they increase when contention occurs for a device. The overall task execution time is determined by a combination of these two costs. However, it is not possible to simply sum the costs, as I/O operations are performed asynchronously (even if the process does not explicitly use asynchronous I/O, kernel and hardware buffering and read-ahead causes many I/O operations to be able to overlap with CPU activity). The task will be able to perform all or some of its CPU operations while an I/O device is simultaneously busy servicing requests for that task.

CPU and I/O costs do not scale the same way, as illustrated in Figure 5.3. As long as the number of parallel tasks does not exceed the number of CPUs, CPU contention is likely to be minimal so CPU costs do not increase significantly. With only a single disk I/O device, those costs increase drastically when contention occurs. Even if both the CPU and disk are under contention, their very nature means that disks are likely to have a larger decrease in performance for each task than the CPU.

In this hypothetical example, the task was CPU bound when running only a single task at a time, but became I/O bound when running two simultaneous



(a) Map tasks (b) Reduce tasks

Figure 5.4: Average CPU and I/O cost of TeraSort tasks

tasks. This kind of effect, where a task that was CPU bound becomes I/O bound due to interference is often observed in practice.

This can be observed happening for real in the TeraSort workload used in Figure 5.1 and Figure 5.2. Figure 5.4 shows how the average CPU and I/O costs of the tasks in the TeraSort workload increase when the number of tasks per node is increased from a single task (no parallelism) to 8 parallel tasks. Because the nodes in the cluster used (see Table 5.4) have 8 CPU cores and the number of parallel tasks never exceeds 8, no significant CPU contention occurs. Therefore, the CPU costs remain stable.

However, the nodes of this cluster only have a single storage device, so this comes under contention when the degree of per-node parallelism is increased. Due to interference between the tasks, the I/O costs for the task increase significantly (the actual costs for each individual task may vary strongly as can be observed from Figure 5.2, Figure 5.4 shows the averages). Just like the hypothetical example from Figure 5.3, the real TeraSort tasks become I/O bound when their I/O costs increase due to interference; the map tasks become I/O bound when more than 5 tasks are executed in parallel, and the reduce tasks become I/O bound with more than 2 parallel tasks (note: this is a simplification; for more details, see Section 5.4).

Although it is possible to reduce the I/O costs by using compression, this often adversely affects the overall performance of the application [CGK10].

It is clear that I/O interference can have a significant and complex effect on the performance of data intensive applications, and that it is necessary to treat I/O costs separately from CPU costs. In order to be able to model the effects of I/O interference accurately, the I/O behavior of data intensive applications must

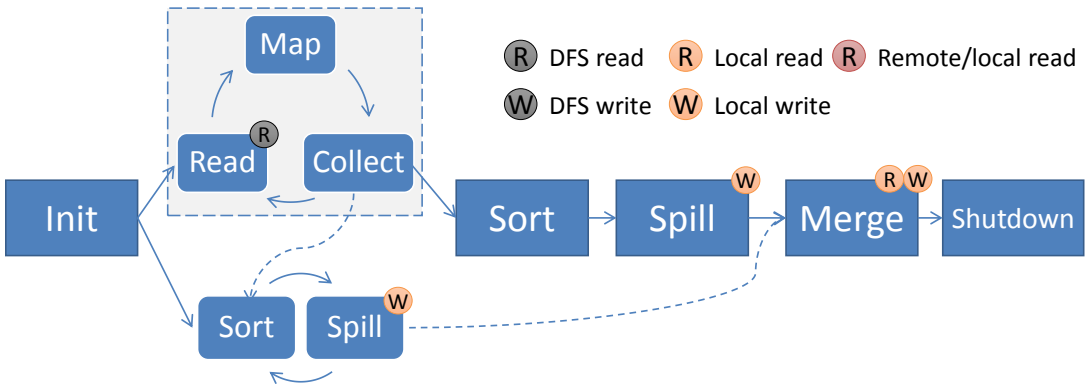


Figure 5.5: Phases of a map task

be considered in more detail.

5.4 I/O behavior of MapReduce

The effects of I/O interference are dependent on the workflow and I/O patterns used by the application. Since the I/O interference model is intended to target MapReduce, its I/O behavior must be examined in detail. Many of these details will vary depending on the implementation used; the descriptions provided in this section are valid for Hadoop 0.20, and should mostly still apply to the Hadoop 0.23/2.0 branch as well as they concern task execution, not scheduling.

5.4.1 Processing workflow

While Section 5.3 discussed CPU and I/O costs in the context of tasks, in reality each task must be further broken down into *phases*: individual steps in the execution of a task. Each phase in a task performs a distinct operation with its own CPU usage and I/O pattern, so each phase has its own cost that must be considered separately from the other phases.

Map tasks

Figure 5.5 shows the structure of a map task in Hadoop, including what types of I/O are performed by each phase (reads or writes on local, remote or DFS storage). The task goes through the following phases:

Init The initialization phase includes starting up the Java Virtual Machine for the task, load task configuration, opening input and output files, and setting

up various streams and objects used by the task. Although this phase may perform some minor I/O operations, primarily on support files (JAR archives and configuration files belonging to the task), this is not significant compared to I/O performed on the actual input and output data by the other phases so this is ignored.

Read Data is read sequentially from the task's input split (typically a block of a file on the Hadoop DFS), and parsed into records to be consumed by the map task. If the input is compressed, decompression is also performed in this phase.

Map The user-specified map function is executed on each record read by the *Read* phase.

Collect Records emitted by the *Map* phase are partitioned and serialized into an in-memory buffer.

Sort When the space utilized by the in-memory buffer reaches a certain threshold, a background thread sorts the contents of the buffer first by partition and then by key. After the map function finishes, the final contents of the buffer are also sorted.

Spill After the buffer has been sorted, its contents are written to a spill file on the local disk. This happens both on a background thread if the buffer reaches the threshold during map execution, and after the map function finishes to spill the final sorted contents of the buffer. If the task uses a combiner function, it will be executed during this function. Spilling may also include compressing the intermediate data.

Merge If more than one spill was performed, the individual spill files must be merged into the task's final output. The merge phase reads all the previous spill files, merges their contents, and writes the output back to local disk (performing compression and decompression as necessary). Depending on the number of spill files, multiple merge passes may be performed. Optionally, the combiner function can be executed again if the number of spills exceeds a user-specified value. If there was only one spill, that spill file is simply renamed to the map task's output file name so no additional I/O is performed.

Shutdown When the task finishes processing, it informs the Tasktracker of successful completion and various execution statistics, and then shuts down any background operations. Depending on the settings for reusing instances of the Java Virtual Machine, the process may be terminated.

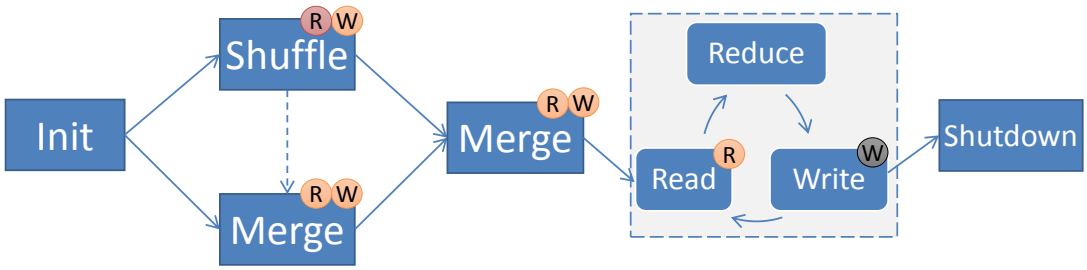


Figure 5.6: Phases of a reduce task

The read, map and collect operations are performed in an interleaved fashion: a single record is read, the processed by the map function and the output for that record is collected before repeating the cycle for the next input record. Because of this, it is considered to be a single phase for the purposes of the I/O interference model in most cases.

Note that the structure of a map task is somewhat different if the job has no reduce tasks. In this case, the *collect* phase is replaced with writing directly to the job output (typically on the DFS) and the *sort*, *spill*, and *merge* phases do not occur. The shutdown phase also commits the output in this case. This situation is trivial so it is not considered further.

All I/O performed by a map task's phases is sequential. The *read* phase sequentially reads a file, and the *spill* phases sequentially write them. The *merge* phase sequentially reads multiple spill files in an interleaved fashion, but aggressive buffering is used so the overhead compared to a regular sequential read is minimal. The output of the *merge* phase is also sequentially written.

Reduce tasks

The structure of a reduce task is shown in Figure 5.6, including information about the I/O operations performed by each phase. The operation of a reduce task goes through three distinct stages—shuffle, sort and reduce—which are made up of the following phases:

Init Reduce task initialization is analogous to map tasks, starting the Java Virtual Machine, loading configuration, and preparing input and output files and other necessary objects for the task. No significant I/O is performed in this phase.

Shuffle During the shuffle phase, the task periodically queries the Tasktrackers on the cluster to see if output files of the map tasks of the job are available. If an output file is available, it is transferred from the remote host into an

in-memory buffer. If a single map output file is too large (by default more than 25% of the total shuffle buffer space), it is shuffled to disk instead, so this phase performs reads of files on remote hosts and can potentially write to files on the local disk.

Merge When the memory buffer reaches a certain threshold, the segments are merged into an on-disk segment by a background thread. If the number of on-disk segments (which includes segments shuffled directly to disk by the *shuffle* phase) exceeds a certain threshold, a background on-disk merge can also be performed.

When the *shuffle* phase finishes, the a portion of the remaining in-memory segments is merged to disk to satisfy the reduce input buffer memory limit (by default this limit is 0%, so all segments are vacated from memory). If the number of on-disk segments is less than the maximum number of disk inputs for a single merge pass (the merge factor), the in-memory segments are vacated by a separate merge operation; otherwise, the in-memory segments are vacated as part of the intermediate merge passes done on the on-disk segments. Intermediate merge passes are performed until the remaining number of on-disk segments is less than or equal to the merge factor.

Read The remaining on-disk segments (and any in-memory segments if not all segments were vacated) are merged here, and the resulting records are passed straight to the *reduce* phase.

Reduce The user-specified reduce function is invoked for each key and the associated set of values.

Write The result of the reduce function is written to a temporary output file on the Hadoop DFS.

Shutdown Reduce task shutdown performs the same operation as map task shutdown, but is also responsible for committing the output. Output is atomically committed by the Jobtracker at which point the temporary output file on the DFS is renamed to the final output file name. This ensures that only the output of a single task attempt is retained if speculative execution was used or a previously thought to have failed task attempt resumes communication after it had already been re-executed elsewhere.

Like the map tasks, most of the I/O performed by the reduce tasks is sequential, with the same comments for the *merge* and *read* phases that applied to the map task's *merge* phase regarding reading multiple streams for merging. The *write* phase also performs strictly sequential writes to the output.

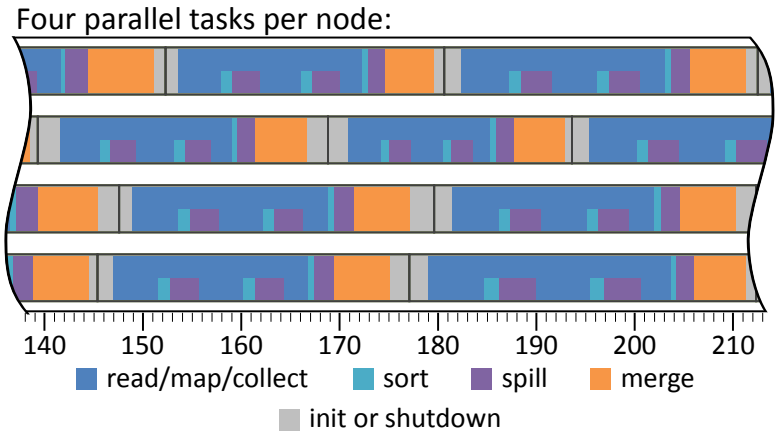


Figure 5.7: Partial time-line of map task phases of the TeraSort workload

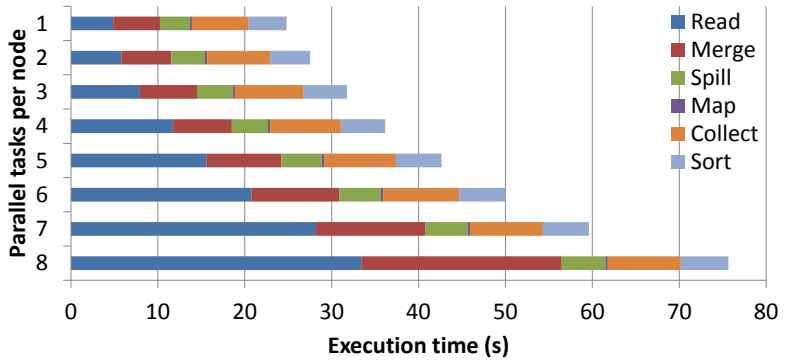
The *shuffle* phase is the only phase that has a significantly different I/O pattern. Reduce task input is divided into *segments*, with one segment being read from the output files of each map task spread over the entire cluster. While each segment itself is read sequentially, the set of segments for a particular reduce task is not stored sequentially (or even on the same node) and is not guaranteed to be read in any particular order. As a result there is a cost for doing a random I/O operation (disk latency and seek costs) involved for each of the segments of a reduce task's input.

5.4.2 I/O interference effects on phases

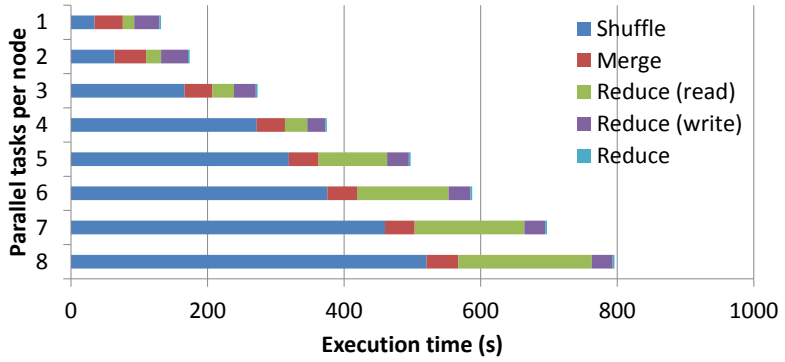
During their execution, map and reduce tasks go through a number of phases of execution. If multiple tasks are running at the same time, it is not guaranteed that all those tasks are executing the same phase at the same time, which can be seen in Figure 5.7. This figure shows a time-line similar to Figure 5.2 amended with information about what phase each task is in at any given time. The half-height bars for *sort* and *spill* indicate that it is processing these phases in the background at the same time as the *read/map/collect* cycle.

The reason that tasks are not always executing the same phase is that tasks are typically not identical in duration (due to data skew, processing skew, or because the tasks are simply part of different workloads), but even for tasks that are normally very similar in execution time this situation can occur because of the variance in task execution time caused by I/O interference that was observed in Figure 5.2.

As a result, a phase may experience interference from any other phase that could potentially be active in other tasks at the same time. The changes in overlap



(a) Map Tasks



(b) Reduce Tasks

Figure 5.8: Execution time of each task phase for TeraSort

are one of the reasons for the observed variability between the tasks (along with scheduler and hardware behavior).

It is not always straightforward to determine which phases have how much interference, because sometimes interference can appear to manifest in a phase other than the one that actually caused it. This can be seen in Figure 5.8, which shows the average execution times of each phase in the TeraSort workload. The figure does not distinguish CPU and I/O cost for each phase because it is not possible to accurately measure those costs for each phase when there is more than one parallel task, partially due to the issue under discussion here. Look for example at the the *spill* phase in Figure 5.8a and the *write* phase in Figure 5.8b. These phases perform large amounts of write I/O and are therefore subject to interference, yet the measured execution time does not appear to increase with higher numbers of parallel tasks. This is because write caching is used on the system, so the measured time only includes the time necessary to write that data into the cache. The time necessary to actually write the data is not entirely gone from the measurements, however, since other tasks (or even the same task) are

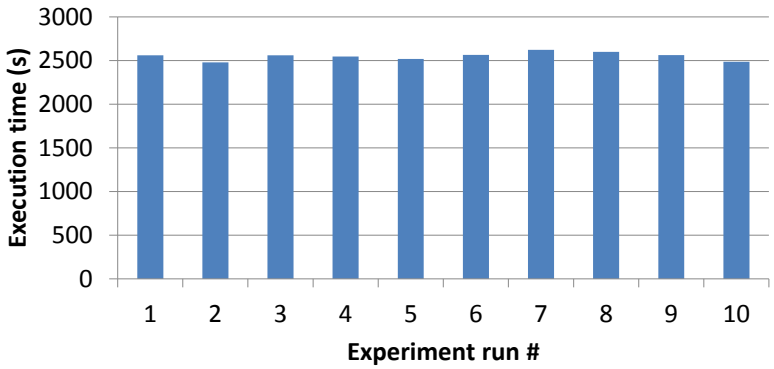


Figure 5.9: Execution times of multiple runs of the TeraSort workload

reading data from disk at the same time, and therefore experiencing interference from those cached writes, increasing their measured execution time.

Because of these issues, it is not possible to predict exactly which phase will be receiving interference from which other phase at which time without an excessively detailed simulation of all components in the system that may affect task execution. Subsequently, it is not possible to give detailed predictions about the behavior of individual tasks in a workload.

However, I observed that when running a workload multiple times the overall execution time of the workload remains relatively stable, even though individual tasks may vary wildly from run to run. Figure 5.9 shows the execution times of 10 individual executions of the TeraSort workload. The overall job execution time varies by at most 3%, and similarly stable values were observed for the average phase execution times like those shown in Figure 5.8. This means that while some phases have more or less interference than others, the overall effect of the interference on the entire workload does not change significantly.

5.4.3 Fragmentation

I/O performance may be affected by external factors that are not directly related to the workload. One such factor is file fragmentation.

Hadoop stores both DFS block files and intermediate files in regular Linux files on the local disks of the nodes in the cluster. This means that Hadoop has no control over the allocation of physical blocks for the files, but is dependent on the kernel I/O scheduler and allocation strategy used by the file system driver. Files used by Hadoop can therefore be subject to file fragmentation depending on the conditions under which they were created. When reading a fragmented file, what appear to be sequential I/O operations from the point of view of the application may in fact not be sequential at all.

Although reading a fragmented file can have a significant impact on performance and be much slower than reading a fully defragmented file, it was not observed to have a significant impact on how processes reading files simultaneously interfere with each other. However, when multiple processes are writing files simultaneously, this has the effect of vastly increasing the amount of fragmentation for those files in many file systems, particularly those that do not support pre-allocation such as the popular ext3 file system (note: Hadoop in particular does not use pre-allocation so even if a file system supports it, it does not improve the situation for Hadoop). Files that were written under conditions of I/O interference may suffer severe fragmentation and therefore reduce performance during reading.

This is particularly important for intermediate files in MapReduce. Intermediate files are written and read by the same job, so any interference during writing affects the performance of reading those files at a later time during the same job. Fragmentation of the intermediate files was observed to have a significant impact on the overall performance of a MapReduce workload in Hadoop in some cases, so being able to predict when fragmentation occurs and how much this will affect subsequent reading of those files is an important part of being able to model MapReduce's I/O behavior.

5.5 I/O cost model

Based on the observations made in the previous sections, I propose an I/O cost model that is able to predict the effects of I/O interference on tasks executed in parallel that are accessing the same disk I/O resource simultaneously.

Figure 5.10 shows the workflow of the prediction process in the I/O interference model. The model consists of two separate components: a hardware interference model and a MapReduce model.

The hardware interference model is used to calculate how the I/O cost of a task changes when I/O interference occurs. This is accomplished by first assuming that bandwidth is split evenly amongst homogeneous tasks, so if there are N tasks running each task gets $\frac{1}{N}$th of the bandwidth, increasing the costs by a factor of N. After this, hardware specific interference functions are applied to the cost. These functions are derived from the hardware environment by running micro-benchmarks for each I/O pattern employed by the application.

The MapReduce model is an analytical model of the MapReduce processing workflow based on the division of tasks into phases according to Figure 5.5 and Figure 5.6. It uses the CPU and I/O cost of a phase, modified to account for interference, to calculate the overall *phase cost*, which essentially represents the execution time of an individual phase. The larger of the two cost values is used,

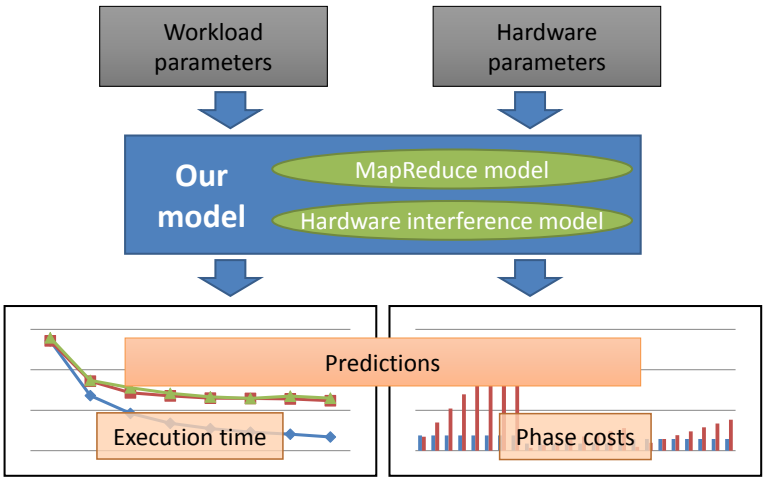


Figure 5.10: Modeling workflow

because I/O operations are performed asynchronously. Based on the phase costs the average *task execution time* is calculated, and from that the map and reduce *stage execution time* and the overall *job execution time*.

The I/O interference model takes as input several parameters, which are given in Table 5.1.

Global workload parameters apply to the workload as a whole rather than individual tasks or phases. Phase parameters have different values for each of the phases of map and reduce tasks, and describe the basic CPU cost of a phase as well as the amount of data read and written by a phase from which the I/O cost can be calculated.

System parameters describe a few configuration parameters of Hadoop that are relevant to the model, which includes the number of parallel map or reduce tasks per node. Hardware parameters provide basic hardware performance characteristics, as well as the hardware specific interference function derived from micro-benchmarks.

Section 5.5.1 describes how workload parameters are estimated, and how hardware parameters and the hardware specific interference functions are determined is explained in Section 5.5.4.

In addition to the parameters from Table 5.1, the equations in this section use additional symbols defined in Table 5.2.

I/O costs can include network transfer costs for I/O operations that are performed on remote hosts. However, in my experiments the disk I/O costs always exceeded the network transfer costs, and since cloud data centers usually provide generous network bandwidth this is expected to often be the case. Therefore, network transfer costs are omitted from the I/O costs in this section.

Workload (global)	
N_{tasks}^{map}	Number of map tasks
N_{tasks}^{reduce}	Number of reduce tasks
Workload (phase)	
S_{read}	Data read in bytes
S_{write}	Data written in bytes
CPU_{phase}	CPU cost in seconds
Environment (system)	
N_{nodes}	The number of nodes in the cluster
N_{slots}^{map}	Number of parallel map tasks per node
N_{slots}^{reduce}	Number of parallel reduce tasks per node
$T_{heartbeat}$	Hadoop heartbeat interval
H_{oob}	Boolean indicating whether out-of-band heartbeats are enabled
Environment (hardware)	
N_{cpus}	The number of CPUs in each node
R_{disk}	The cost of reading 1 byte, in seconds
W_{disk}	The cost of writing 1 byte, in seconds
C_{random}	The cost of a random I/O operation
F	The fragmentation factor
$F_{threshold}^{shuffle}$	The threshold segment size for fragmentation in the shuffle phase
f_i	The hardware specific interference function

Table 5.1: Model parameters

The goal of the model is to predict how performance is affected when the system parameters N_{slots}^{map} and N_{slots}^{reduce} change, which express the degree of parallelism per node for both types of tasks. These values can be configured separately, and indeed often have different optimal values even for the same workload.

The output of the model is a prediction about the effect of interference for each value of N_{slots}^{map} and N_{slots}^{reduce} on the average map and reduce task execution time respectively, as well as the overall job execution time; it also provides predictions about the costs of each phase that provide insights into which parts of the application are I/O bottlenecks.

Symbol	Meaning
CPU_{phase}^{actual}	CPU cost of a phase including CPU contention
IO_{phase}^{read}	Read I/O cost of a phase including I/O interference
IO_{phase}^{write}	Write I/O cost of a phase including I/O interference
C_{phase}	The total cost of a phase (used as a placeholder for the specific phases given below)
C_{init}^{map}	Map task init phase cost
C_{rmc}^{map}	Map task read/map/collect phases cost
C_{sort}^{map}	Map task sort phase cost
C_{spill}^{map}	Map task spill phase cost
C_{merge}^{map}	Map task merge phase cost
C_{init}^{reduce}	Reduce task init phase cost
$C_{shuffle}^{reduce}$	Reduce task shuffle phase cost
C_{merge}^{reduce}	Reduce task merge phase cost
C_{rrw}^{reduce}	Reduce task read/reduce/write phases cost
T_{work}^{map}	Total execution time of all processing phases in a map task
T_{work}^{reduce}	Total execution time of all processing phases in a reduce task
$T_{interval}^{poll}$	Shutdown polling interval
T_{task}^{map}	Total execution time of a map task
T_{task}^{reduce}	Total execution time of a reduce task
T_{slot}^{map}	The time a map task occupies a slot
T_{slot}^{reduce}	The time a reduce task occupies a slot
T_{stage}^{map}	Map stage total execution time
T_{stage}^{reduce}	Reduce stage total execution time
T_{job}	The total time for the workload
$R_{throughput}$	Average hard disk read throughput
$W_{throughput}$	Average hard disk write throughput
$R_{throughput}^{fragmented}$	Throughput of reading a fragmented file
N_{slots}	Placeholder for either N_{slots}^{map} or N_{slots}^{reduce}, depending on context

Table 5.2: Additional model symbols

5.5.1 Measuring workload parameters

The workload parameters are given in Table 5.1. The parameters are determined from two primary sources, job history and log files. When executing a job, the Jobtracker records a history of events for the job which includes the start and end time of each task attempt, and the values of all counters for each task attempt (counters contain information such as the number of records and the amount of DFS and local data read and written). During execution, each task creates a log file containing diagnostic information about the execution of that task. Both job history files and task log files are retained by Hadoop for some time after a job finishes (even if the cluster is restarted), so they are available to read the information from.

Global workload parameters are trivially determined, as they depend only on the workload's input data and the configuration used by the user. N_{tasks}^{map} is determined by the number of DFS blocks the input has and the maximum split size. N_{slots}^{reduce} must be explicitly set by either the system administrator or the user that submits the workload. If a workload is already executed, these parameters can be trivially determined from the job history file created by Hadoop.

Phase parameters need to be determined per phase, and the only way to determine them is to execute the tasks in the workload. For each phase, the values of CPU_{cost}, S_{read} and S_{write} need to be determined. For some phases, the values of S_{read} and S_{write} are already recorded in the task's counters or its log file, but in some cases the information is either not recorded or combines the figures from several phases (for example, each task maintains a counter of how many local bytes were read and written, but it does not indicate how many of those were written by e.g. the *spill* or *merge* phases for a map task).

In order to be able to determine these parameters, I made two modifications to Hadoop 0.20.203.0:

1. Timers are kept during execution to record the CPU time and elapsed time of the phases. This information is recorded in the task log files.

2. Read and write data sizes for phases that were not already recorded in either the log file or history file are added to the log file.

No further modifications are made to Hadoop. I considered using a profiler instead of modifying Hadoop similar to the approach used in [HDB11], but found that the overhead of the profiler skewed the results too much.

In order to save time, it is not necessary to execute the entire workload. Instead, only a portion of the tasks need to be sampled to get a reasonably accurate value for the average costs.

After the test workload is executed, an automated tool retrieves and processes the job history, task log files, Datanode log files, and Tasktracker log files, extracting the values of the parameters for each phase.

The modified version of Hadoop and the workload analyzer are described in further detail in Section 6.4.

5.5.2 Phase cost estimation

Each phase has its own cost, which has its own symbol in the model. The symbols for the phases are C^{map}_{init}, C^{map}_{rmc}, C^{map}_{sort}, C^{map}_{spill}, C^{map}_{merge}, C^{reduce}_{init}, $C^{reduce}_{shuffle}$, C^{reduce}_{merge}, and C^{reduce}_{rrw}. Note that the shutdown phase for both map and reduce tasks is treated specially and does not have a normal cost calculation; how the shutdown phase is factored in is explained in Section 5.5.3.

Because the formulas for each phase are largely the same, the symbol C_{phase} is used as a generic symbol indicating the cost of any of those phases. It should be replaced with any of the above symbols depending on which phase is being calculated. Similarly, values of phase-specific parameters and symbols such as CPU_{phase}, S_{read} and S_{write} should be set to their respective values for the relevant phase. The symbol N_{slots} is used to represent either N^{map}_{slots} or N^{reduce}_{slots} depending on whether the phase is part of a map or reduce task respectively.

The basic cost calculation used for each phase is as follows:

$$C_{phase} = \max(CPU^{actual}_{phase}, IO^{read}_{phase} + IO^{write}_{phase}) + f_i \tag{5.1}$$

Phase costs are calculated by taking the higher value of either the CPU cost or I/O cost for a phase, based on the observation that I/O is performed asynchronously. The hardware specific interference function f_i, which is described in Section 5.5.4, is added separately from the I/O costs because it may interact differently with the CPU costs in some environments (for example, some interference costs may not be able to fully overlap with any CPU costs, such as interference that happens when a stream first starts reading); if this is the case it should be reflected by f_i itself.

CPU^{actual}_{phase} is a modified version of CPU_{phase} that accounts for any CPU contention that may occur when running tasks in parallel. It is calculated under the assumption that a phase can at most use one core (the phases themselves do not contain parallelism), and that CPU time is divided equally between tasks if the number of tasks exceeds the number of CPU cores:

$$CPU^{actual}_{phase} = CPU_{phase} \cdot \max(1, \frac{N_{slots}}{N_{cpus}}) \tag{5.2}$$

This equation increases the CPU costs if the number of parallel tasks exceeds the number of costs, but does not decrease it if there are unused CPU cores due to the fact that a task can use at most one CPU core.

Equation (5.1) and Equation (5.2) are used for all phases, but the calculation of IO^{read}_{phase} and IO^{write}_{phase} is dependent on the IO pattern used by the phase. The calculations for each of the I/O patterns for MapReduce are given below.

Sequential I/O

For phases that perform sequential I/O operations, the basic I/O costs for reading and writing are calculated by simply multiplying the data sizes with the hardware cost of reading and writing:

$$IO^{read}_{phase} = N_{slots} \cdot S_{read} \cdot R_{disk} \tag{5.3}$$

$$IO^{write}_{phase} = N_{slots} \cdot S_{write} \cdot W_{disk} \tag{5.4}$$

The basic I/O costs are multiplied by the value of N_{slots} to represent the assumption that bandwidth is divided evenly between tasks running simultaneously. Additional interference between the parallel streams is incorporated by f_i, and not by these I/O cost calculations.

Equation (5.3) and Equation (5.4) are used for the values of IO^{read}_{phase} and IO^{write}_{phase} in Equation (5.1) for the calculation of the costs for the map task *init* (C^{map}_{init}), *read/ map/collect* (C^{map}_{rmc}), *sort* (C^{map}_{sort}), *spill* (C^{map}_{spill}), and *merge* (C^{map}_{merge}) phases, and the reduce task *init* (C^{reduce}_{init}) and *merge* (C^{reduce}_{merge}) phases.

Note that in several of these phases, the value of IO^{read}_{phase} or IO^{write}_{phase} can be zero, because not all phases perform read or write I/O. For the map and reduce task *init* phases, and the map task *sort* phase no I/O is performed at all, so the value of C^{map}_{init}, C^{map}_{sort} and C^{reduce}_{init} is always equal to CPU^{actual}_{phase}.

Fragmentation

As noted in Section 5.4.3, intermediate files may become fragmented when multiple tasks are writing intermediate files concurrently. As a result of that fragmentation, the performance of reading those files back may be considerably lower. Therefore, while the cost of writing the files does not change, it is necessary to alter the cost of reading the files for any phase that reads intermediate files that may have been fragmented by interference when they were written. In theory, this can happen to any phase that reads intermediate files. In practice, I only found this to have a significant effect on the reduce task *shuffle* and *read* phases.

Intuitively, fragmentation will cause an additional non-linear decrease of read throughput (or a non-linear increase of I/O costs) when the degree of parallelism is increased. The size of this additional cost increase depends on the amount of fragmentation caused by the interference. To express this level of fragmentation,

I use the fragmentation factor parameter F whose derivation is described in Section 5.5.4. For phases where fragmentation is an issue, IO_{phase}^{read} is replaced with $IO_{phase}^{read\ \prime}$:

$$IO_{phase}^{read\ \prime} = {N_{slots}}^{F} \cdot S_{read} \cdot R_{disk} \tag{5.5}$$

The value $IO_{phase}^{read\ \prime}$ from Equation (5.5) is used in Equation (5.1) for the calculation of the I/O costs of the reduce task *read/reduce/write* phase (C_{rrw}^{reduce}). Equation (5.4) is used for IO_{phase}^{write}, because all writes performed by the reduce task *write* phase are sequential.

Shuffle I/O

For the I/O pattern used by the reduce task *shuffle* phase, additional costs for the random I/O operations performed for each input segment must be added to the phase cost. Because I found that these costs cannot overlap with the CPU costs, the value of C_{phase} from Equation (5.1) is augmented as follows:

$${C_{phase}}' = C_{phase} + N_{slots} \cdot (N_{tasks}^{map} \cdot C_{random}) \tag{5.6}$$

Each reduce task reads a single segment from each map task, so the number of segments is equal to the number of map tasks (N_{tasks}^{map}). The cost of the random I/O (C_{random}) is hardware specific and further explained in Section 5.5.4.

The shuffle phase may be subject to fragmentation, but I found this did not manifest for small segment sizes; if the segments are very small, the odds of a non-contiguous read occurring within a segment decrease and the effect of fragmentation will be negligible. In the model this is expressed using $F_{threshold}^{shuffle}$, which is the minimum segment size before fragmentation is assumed to take effect. Therefore, IO_{phase}^{read} from Equation (5.1) is replaced with the alternative value $IO_{shuffle}^{read}$:

$$IO_{shuffle}^{read} = \begin{cases} IO_{phase}^{read} & \text{if } \frac{S_{read}}{N_{tasks}^{map}} < F_{threshold}^{shuffle} \\ IO_{phase}^{read\ \prime} & \text{if } \frac{S_{read}}{N_{tasks}^{map}} >= F_{threshold}^{shuffle} \end{cases} \tag{5.7}$$

The value ${C_{phase}}'$ from Equation (5.6) using $IO_{shuffle}^{read}$ is used for the calculation of $C_{shuffle}^{reduce}$. The value of IO_{phase}^{write} is the same as for sequential I/O.

Remote I/O

When a task performs I/O on a remote host, it contributes to the interference on the destination node rather than the one the task is running on. This can occur for map tasks when they are scheduled on a node that does not have the task's input data. However, the scheduler tries to avoid this and it does not happen often enough to significantly affect the execution time.

For reduce tasks, the shuffle phase reads data from both local and remote hosts. It is necessary to know how much data is read on average from each node per task. Each task reads on average $\frac{1}{N_{nodes}} \cdot S_{read}$ bytes from each node (including its local node), so the total amount of data read per node is $\frac{N_{tasks}}{N_{nodes}} \cdot S_{read}$. To get the average per task, this value is divided by the number of tasks per node, $\frac{N_{tasks}}{N_{nodes}}$, which equals S_{read}. This means no adjustment to the shuffle phase formula is necessary.

Read and write caching

The input data for any phase may already be cached by for example the operating system page cache or any hardware caches being used. If this is the case the I/O costs of a phase are reduced depending on the amount of data that did not need to be read from the disk.

For the input data of the job it cannot be known whether this data is cached as the circumstances and time of its creation are usually not known. Therefore, the input of the map tasks is always assumed to be entirely not cached. During the experimental evaluation in Section 5.6, the cache is always cleared before each experiment so that this is the case.

Intermediate data is more complex. This data is written by the job, and parts of it may still reside in memory by the time it is read back. Whether or not this is the case and how much data is retained depends on the size of the data, the time between writing and reading back, the size of node's cache memory, memory and cache utilization of the system (both from this workload and any other processes on the system), etc. Accurately predicting when caching occurs and by how much is therefore very complicated and outside of the scope of this work.

To deal with caching, I use a simple heuristic that says that if the data size per task and the total data size in a wave of tasks on a single node is below a certain threshold for some phases (the map task *merge* phase, the *shuffle* phase, and the reduce task *read* phase), all data is assumed to be cached. The main reason for this is the shuffle phase: if the intermediate data is very small, the random I/O cost per segment dominates the cost of the shuffle phase, but this cost may not be incurred if the data is cached leading to a very large over-estimation. By assuming the data is cached for very small data sizes, this overhead is avoided. On the environment described in Table 5.4, the task threshold is set at 12.5% of main memory and the wave threshold at 25% of memory. For the shuffle phase, the total output size of all map tasks per node must not exceed 25% of main memory. When these conditions are met, S_{read} for that phase is considered 0.

Writing is also a complex issue. All write operations are normally cached and flushed to disk at a later time. Therefore, the actual phase doing the writing may not observe any of the I/O costs of writing. However, when the data is written

to the device this may slow down other parts of the workload, which is why these costs must still be included.

Flushing is handled by the file system driver, which regularly flushes data, usually preferring to do this when the device is not fully utilized. However, when the amount of dirty pages in the cache exceeds the `dirty_background_ratio` parameter of the Linux kernel, flushes will be scheduled immediately. These writes may occur during unfavorable conditions, and can block read operations on the same device for a significant amount of time. When the amount of dirty pages exceeds `dirty_ratio`, processes will be forced to perform write-back during their time slot. Note that any process can be forced to do this; it is not necessarily the process that caused the high write pressure, or even one that is writing to the same device.

Because of these factors, if the amount of data being written is very large writes essentially become synchronous. As with read caching, it is very difficult to accurately predict when this situation will occur. Therefore, I again use a heuristic approach such that if the total data size read and written in a task is above a certain threshold and the total data size of the wave is above a certain threshold, writes are considered to be synchronous and can no longer overlap with CPU time. This is used for the reduce task *read/reduce/write* phase, and for the combined *shuffle* and *merge* phases. On the environment described in Table 5.4, the task threshold is set at 25% of main memory and the wave threshold at 50% of memory.

5.5.3 Task and workload estimation

Once the individual phase costs are determined, the overall execution time of a map or reduce task can basically be estimated as a sum of the individual phase costs. Although some phases may be executed in parallel with other phases (for example the map task *sort* and *spill* phases can overlap with the execution of the *read/map/reduce* phases), I use the assumption that either the node's CPU cores are fully saturated or the tasks are I/O limited, so in either case there is no practical difference to using a sum.

However, one phase needs to be given special consideration, because it does not perform any real computation: the *shutdown* phase for both map and reduce tasks. The execution time of the *shutdown* phase is determined primarily by a waiting operation that depends on a a polling interval. Basically, the *shutdown* phase waits for a background progress reporting thread to end, and this thread checks whether it needs to end on a set interval, represented here as $T^{poll}_{interval}$ (this value is hard coded at three seconds in Hadoop 0.20). This thread is started at the end of the *init* phase for both map and reduce tasks, so that is the time from which polling starts.

In order to account for this behavior of the *shutdown* phase, I first determine the *work time* of the map and reduce tasks, which is the time spent in all phases that perform actual work (all phases except *init* and *shutdown*. For map tasks, this is calculated as follows:

$$T_{work}^{map} = C_{rmc}^{map} + C_{sort}^{map} + C_{spill}^{map} + C_{merge}^{map} \tag{5.8}$$

And analogously for reduce tasks:

$$T_{work}^{reduce} = C_{shuffle}^{reduce} + C_{merge}^{reduce} + C_{rrw}^{reduce} \tag{5.9}$$

To calculate the total execution time for a task, this value must be rounded up to the nearest multiple of the shutdown polling interval $T_{interval}^{poll}$, and the cost of the *init* phase must be added. For map tasks:

$$T_{task}^{map} = C_{init}^{map} + \left\lceil \frac{T_{work}^{map}}{T_{interval}^{poll}} \right\rceil \cdot T_{interval}^{poll} \tag{5.10}$$

And similarly for reduce tasks

$$T_{task}^{reduce} = C_{init}^{reduce} + \left\lceil \frac{T_{work}^{reduce}}{T_{interval}^{poll}} \right\rceil \cdot T_{interval}^{poll} \tag{5.11}$$

This is however not the final calculation done for a task. The Jobtracker only schedules a new task for a newly available slot when it receives a heartbeat from the relevant Jobtracker. Therefore, the time that a task actually occupies a slot lasts until the next heartbeat after the task finishes. Since tasks are always started on heartbeats, this means the task execution time must be rounded up to the nearest multiple of the heartbeat interval (which is calculated based on the number of nodes but is at least 3 seconds in Hadoop 0.20).

Hadoop can be configured to use *out-of-band heartbeats*. If this option is enabled, a Tasktracker sends a heartbeat to the Jobtracker immediately after a task finishes. This means there is no delay in scheduling new tasks at the cost of increased communication between the Tasktrackers and the Jobtracker. In the model, the system parameter H_{oob} indicates whether out-of-band heartbeats are enabled or not.

Therefore, the calculation of the slot time is as follows for map tasks:

$$T_{slot}^{map} = \begin{cases} \left\lceil \frac{T_{task}^{map}}{T_{heartbeat}} \right\rceil \cdot T_{heartbeat} & \text{if } \neg H_{oob} \\ T_{task}^{map} & \text{if } H_{oob} \end{cases} \tag{5.12}$$

And for reduce tasks:

$$T_{slot}^{reduce} = \begin{cases} \left\lceil \frac{T_{task}^{reduce}}{T_{heartbeat}} \right\rceil \cdot T_{heartbeat} & \text{if } \neg H_{oob} \\ T_{task}^{reduce} & \text{if } H_{oob} \end{cases} \tag{5.13}$$

Based on the slot time for a task, the overall execution time of the map and reduce stages, T_{stage}^{map} and T_{stage}^{reduce}, can be calculated. Each stage executes in a number of waves, where each wave consists of the total number of tasks the cluster can run in parallel for that task type (the cluster capacity). At the end of each stage, there may not be enough tasks left in the final wave so that all nodes are running N_{slots} tasks. During this final wave there would be less interference on those nodes than what the model predicts. Because this is outside the target of this model, the last wave is excluded from the stage time. If desired, a separate interference calculation could be done for the final wave and that value could be added to the stage times, but task skew, stragglers, and the fact that the scheduler is not guaranteed to run the same number of tasks on each node in these circumstances mean that this calculation would necessarily be less accurate than the calculation for the "full" waves.

The map stage execution time is calculated as follows:

$$T_{stage}^{map} = \left\lfloor \frac{N_{tasks}^{map}}{N_{slots}^{map} \cdot N_{nodes}} \right\rfloor \cdot T_{slot}^{map} \tag{5.14}$$

I take the floor of the number of waves to represent the fact that the final, partial wave is excluded.

The reduce stage execution time is calculated as follows:

$$T_{stage}^{reduce} = \left\lfloor \frac{N_{tasks}^{reduc}}{N_{slots}^{map} \cdot N_{nodes}} \right\rfloor \cdot T_{slot}^{reduce} \tag{5.15}$$

The total execution time of the job can then be estimated as follows (with the caveat that the real execution time is slightly longer because of the final wave):

$$T_{job} = T_{stage}^{map} + T_{stage}^{reduce} \tag{5.16}$$

Although it is possible for the first wave of reduce tasks to be scheduled while the map tasks of the job are still running, in practice I found this did not affect the overall execution time of the job much. Additionally, the Hadoop YARN scheduler in Hadoop 0.23 (the current alpha version of Hadoop) removes the notion of slots and the distinction between map and reduce tasks, instead scheduling based on resource requests. In Hadoop YARN, it therefore no longer occurs that a reduce task for a job can be scheduled while there are still remaining unscheduled map tasks.

5.5.4 I/O interference estimation

I/O interference and the effects it has depend on a large number of factors, including application behavior, the operating system kernel scheduler and I/O scheduler, hardware behavior, and others. Because it would not be feasible to create a detailed analytical model of all these components for any given environment, instead I use a black-box approach to approximate the behavior of the hardware environment.

Micro-benchmarks are used for each I/O pattern to determine the hardware specific interference function and other relevant hardware parameters given in Table 5.1. These benchmarks are used to observe how the system behaves under circumstances that replicate those found in the actual application. The micro-benchmarks do not attempt to emulate the overall application behavior; each focuses on only a single I/O pattern, and the MapReduce model described above is used to tie them together.

In this section, the results of each benchmark and the functions and values derived from them are given. These results are based on the hardware environment described in Table 5.4. However, the method used to finding these values is applicable to any environment.

Because the systems used for these experiments have 8 CPU cores and I do not intend to run more than 8 tasks in parallel on them, the maximum degree of parallelism employed in the benchmarks is also set at 8.

The following sections provide the benchmarks and their results for the I/O patterns used by Hadoop.

Sequential I/O

For sequential I/O operations, the values of the R_{disk} and W_{disk} parameters must be determined, as well as the hardware dependent interference function f_i.

The R_{disk} and W_{disk} parameters represent the cost of reading from or writing to a disk, in seconds per byte. This cost is the calculated using the inverse of the throughput in bytes per second:

$$R_{disk} = \frac{1}{R_{throughput}} \tag{5.17}$$

$$W_{disk} = \frac{1}{W_{throughput}} \tag{5.18}$$

$R_{throughput}$ and $W_{throughput}$ are the average read and write throughput for the device, measured using a single stream. To adjust the value as necessary for larger number of streams, f_i is used.

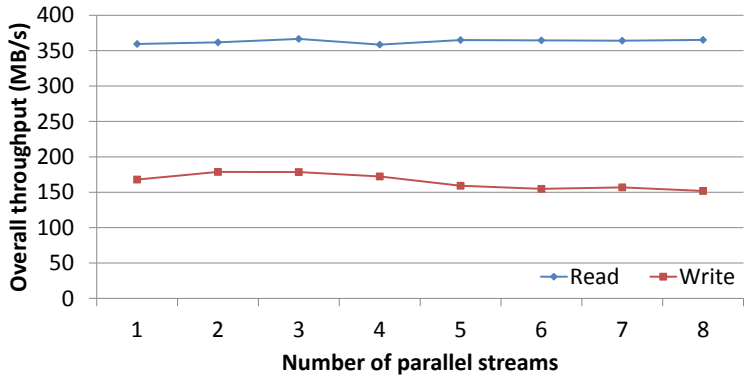


Figure 5.11: Read and write throughput of the hardware environment from Table 5.4, using between 1 and 8 parallel streams

To determine the values of these parameters, I use a micro-benchmark that does the following:

- Before each experiment, the operating system cache is cleared using a kernel software option, and any hardware caches are cleared by reading 2 gigabytes of unrelated data.
- Read N files in parallel from the disk, and record a time-line of the operation.
- Write N files in parallel from the disk, and record a time-line of the operation. The `O_SYNC` option is used to bypass any operating system and hardware write caching.

The value of N is varied to get information about different degrees of parallelism (in this case, it is varied between 1 and 8).

Figure 5.11 shows the read and write throughput measured by the micro-benchmark for each number of parallel streams. The throughput value shown is the aggregate throughput across all streams.

For reading, I observe an average throughput of 360 megabytes per second for a single stream, and the total throughput does not diminish significantly when the number of parallel streams is increased. While each individual stream of course gets a smaller portion of the overall throughput, the total performance of the device is not significantly affected. This is because each node in this environment uses a RAID array for storage, which has a large (1GB) cache and uses an efficient read-ahead caching strategy for multiple sequential streams.

However, the time-line of reading the streams must also be considered. A partial time-line for reading a single stream is shown in Figure 5.12. This figure shows how the throughput of the stream changes over time as it is being read.

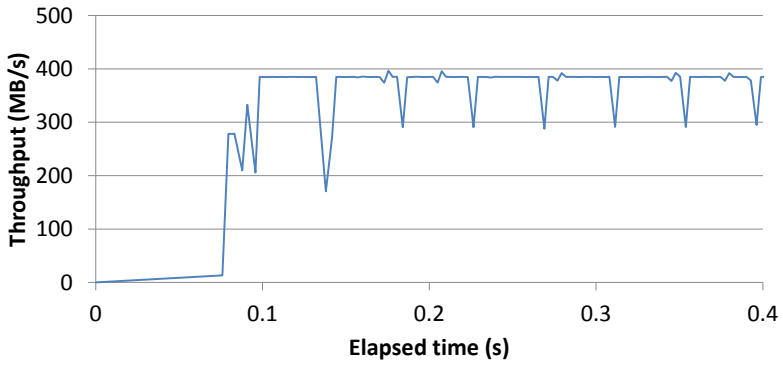


Figure 5.12: Time-line of read throughput when reading a single stream on the hardware environment from Table 5.4

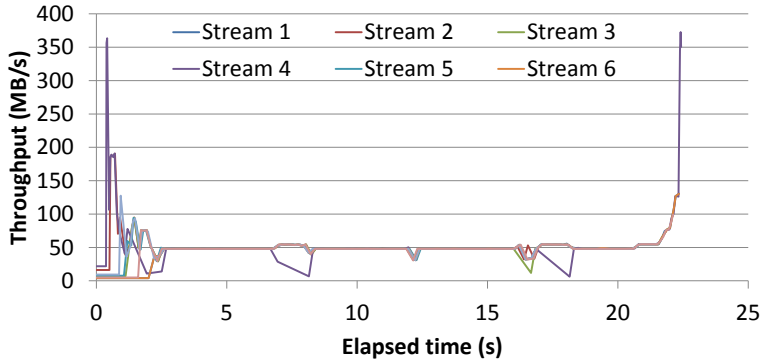


Figure 5.13: Time-line of read throughput when reading 8 streams in parallel

What can be observed is that there is a delay of almost 100 milliseconds before reaching a stable throughput of approximately 384 megabytes per second. This delay at the start is caused by random I/O costs to seek to the start of the stream, and a delay before the RAID array starts read-ahead caching. A small drop in throughput is observed every 50 milliseconds or so, which I believe is also due to the size of the read-ahead cache (the dips occur almost exactly every 16 megabytes into the file after the start of read-ahead).

A time-line for reading 8 streams in parallel is shown in Figure 5.13. What can be observed is that the streams initially perform very erratically, as they no longer all show the same delay at the start. This increased delay is caused by interference between the streams before the read-ahead kicks in. However, when all streams have properly started reading, they are able to each maintain the same stable throughput, and share the device's total throughput between them equally.

The increase in the start-up delay due to the interference before read-ahead is

somewhat erratic, but on average it appears to be increase linearly in respect to the number of parallel streams.

Based on these observations, the value for $R_{throughput}$ is set at 380 megabytes per second (slightly lower than the observed peak throughput to account for the occasional drops). This means that R_{disk} is set as follows:

$$R_{disk} = \frac{1}{398458880} = 2.5 \times 10^{-9} \tag{5.19}$$

To account for the 100 millisecond start-up delay and the increase of that delay when the number of streams increases, I determine f_i to be as follows for reads:

$$f_i = \begin{cases} 0.1 \cdot N_{slots} & \text{if } S_{read} > 0 \\ 0 & \text{if } S_{read} = 0 \end{cases} \tag{5.20}$$

For writing, the average throughput for a single stream seen in Figure 5.11 is approximately 167 megabytes per second. Writing performance is likely lower than reading performance due to the use of a RAID6 volume, which adds additional overhead for writing due to the necessary updates on the checksum volumes. When the number of parallel streams is increased, the aggregate throughput across all streams initially remains stable but then drops off to about 150 megabytes per second.

Unlike with reading, the time-line for writing shows no delay at the start, and all streams have a stable throughput throughout regardless of the number of parallel streams. Therefore, the observed average throughput can be used for $W_{throughput}$. To ease the approximation of the interference, the lowest observed total through of 150 megabytes per second is used. This means that W_{disk} is set as follows:

$$W_{disk} = \frac{1}{157286400} = 6.4 \times 10^{-9} \tag{5.21}$$

It is not necessary to modify f_i for writes.

For some devices (for example regular single hard disks), behavior when reading multiple streams could be much more complicated, which means it is much more difficult to determine f_i. It may be necessary to employ statistical approximations to determine an accurate interference function in this case. Refining the technique for finding f_i for different environments is part of the future work for this research.

Shuffle I/O

The reduce task *shuffle* phase reads input data in segments, and it is necessary to measure the random I/O cost associated with each segment to determine the value of C_{random} for use in Equation (5.6).

The micro-benchmark used to determine these values works as follows:

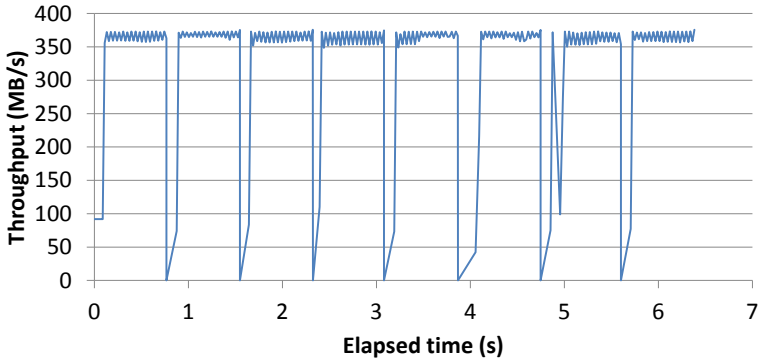


Figure 5.14: Time-line of read throughput reading segments from multiple files using a 256 megabytes segment size

- Create 256 files containing random data, from which the segments will be read.
- Before each experiment, the operating system cache is cleared using a kernel software option, and any hardware caches are cleared by reading 2 gigabytes of unrelated data.
- Read a total of 2 gigabytes by reading S megabytes from each file, starting at a random offset in each file. Record a time-line of the operation.

The value of the segment size S is varied between 8 and 256 megabytes. Because the total size of the data read is kept at 2 gigabytes, this has the effect of changing the number of segments read. The full 256 input files are only used with the 8 megabytes segment size.

Figure 5.14 shows a partial time-line when the segment size is 256 megabytes is used on the hardware environment from Table 5.4. It can be observed that the delay between each segment matches the 100 millisecond delay at the start of each stream when measuring sequential I/O. When the segment size is decreased, this observation holds. Therefore:

$$C_{random} = 0.1 \tag{5.22}$$

In order to test whether this approximation works before applying it to the model as a whole, I compared the transfer time taken from the micro-benchmark against a prediction using this value of C_{random}. The transfer time was predicted using the following equation derived from Equation (5.6):

$$S_{read} \cdot R_{disk} + \frac{S_{read}}{S_{segment}} \cdot C_{random} \tag{5.23}$$

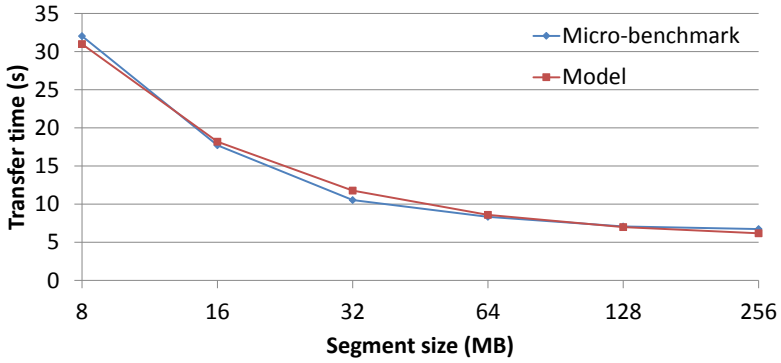


Figure 5.15: Measured and predicted shuffle I/O performance

S_{read} is the 2 gigabytes used by the micro-benchmark, and $S_{segment}$ is the segment size. The results are shown in Figure 5.15, which indicates that the prediction corresponds almost perfectly with the observed values.

Fragmentation

To determine the effect of fragmentation, the performance of reading a file that was written under interference conditions needs to be measured. To do this, the following micro-benchmark was used:

- Before each experiment, the operating system cache is cleared using a kernel software option, and any hardware caches are cleared by reading 2 gigabytes of unrelated data.
- Write N files in parallel, writing 1 gigabyte of data to each file.
- Read back the created files sequentially, averaging the read throughput.

This measures the value of $R_{throughput}^{fragmented}$ for a particular level of fragmentation created by using a given number of parallel writers. The number of parallel writers N is once again varied between 1 and 8.

In order to find the fragmentation factor F needed for Equation (5.5), the following formula has to be solved for F for values of $N > 1$:

$$N^{F-1} = \frac{1}{R_{disk} \cdot R_{throughput}^{fragmented}} \tag{5.24}$$

This gives a different value of F for each value of N, but the values were very close together as seen in Table 5.3 so the average is a reasonable approximation. Therefore, $F = 1.74$ for the environment described in Table 5.4.

Writing streams	$R_{throughput}^{fragmented}$ (MB/s)	F
2	235.75	1.69
3	176.786	1.7
4	139.751	1.72
5	116.17	1.74
6	98.656	1.75
7	89.174	1.74
8	78.7059	1.76

Table 5.3: Throughput of reading files created with interference during writing and the corresponding fragmentation factors

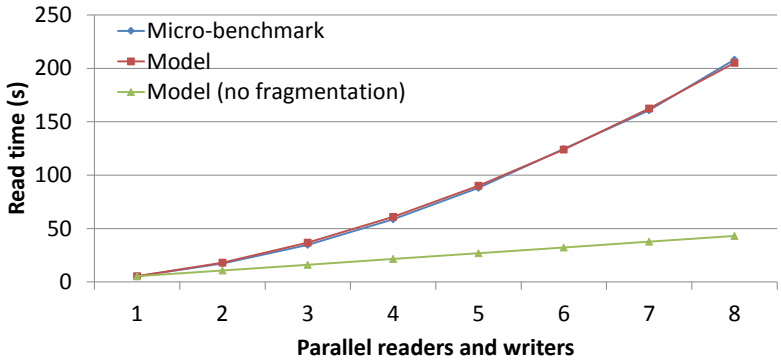


Figure 5.16: Measured and predicted read performance with fragmentation

This approximation was evaluated against the micro-benchmark by using the determined value of F to predict the transfer time of the micro-benchmark using the following equation derived from Equation (5.5):

$$N_{streams}{}^{F} \cdot S_{read} \cdot R_{disk} \tag{5.25}$$

Figure 5.16 shows the result of the comparison. The predicted value matches the measured value very closely for each number of streams. Also shown in this figure is what the predicted performance would be if fragmentation was not taken into account. The difference between the two predictions gets very big for larger number of streams used to create the files, so obviously fragmentation has a serious impact on performance when it occurs.

For fragmentation in the *shuffle* phase, I must also determine the value of $F_{threshold}^{shuffle}$. To determine this, the micro-benchmark for shuffle I/O is re-executed but using input files that were generated using varying numbers of parallel writers. Figure 5.17 compares the prediction using Equation (5.23) (which does not account for fragmentation) to the measured values with fragmentation created by using 4

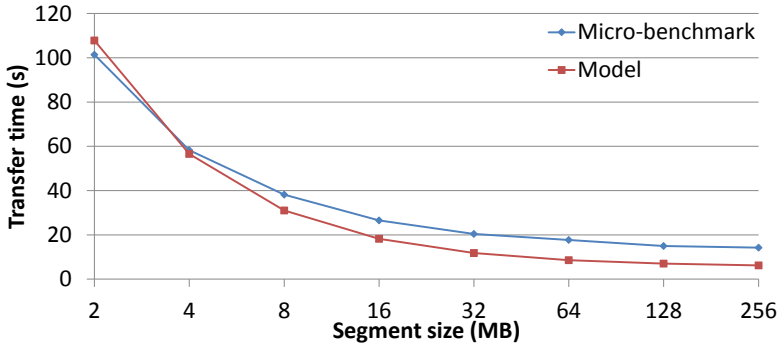


Figure 5.17: Measured and predicted shuffle I/O performance with fragmentation

System	
CPU	2x quad-core Xeon E5530 2.4GHz
Memory	24GB
Storage (RAID)	
Controller	JCS VCRVAX-4F RAID
HBA	QLogic QLE2462
Disks	10x Hitachi HDS721010CLA332

Table 5.4: Experimental environment

parallel writers. It can be seen that the transfer time begins to increase at around 8MB segment size, and this is confirmed by the results using other numbers of parallel writers. Therefore:

$$F_{threshold}^{shuffle} = 8388608 \tag{5.26}$$

5.6 Experimental evaluation

I evaluated the model by running several workloads on a 10 node cluster using Hadoop 0.20.203.0. Table 5.4 shows the hardware configuration of each node. I obtained the parameters given in Table 5.1 for the hardware and for each workload using the methods given in Section 5.5. The workloads were then executed normally, varying the number of map and reduce tasks per node between 1 and 8, and the actual performance was compared to the predictions made by the model.

The map stage and reduce stage are evaluated separately because this provides more interesting insights than just the overall job execution time. To facilitate this, Hadoop was configured to delay reduce task scheduling until all map tasks

Parameter	Value
`dfs.block.size`	128 MB
`dfs.replication`	3
`mapred.child.java.opts`	-Xmx2048M
`mapred.map.tasks.speculative.execution`	false
`mapred.reduce.tasks.speculative.execution`	false
`mapred.reduce.parallel.copies`	1
`mapred.reduce.slowstart.completed.maps`	1.0
`mapreduce.tasktracker.outofband.heartbeat`	true

Table 5.5: Hadoop configuration

		TeraSort	WordCount	PFP Growth	PageRank
Data size	**Input**	160 GB	40 GB	45 GB	120 GB
	Intermediate	160 GB	100 MB	240 GB	370 GB
	Output	160 GB	1 MB	10 MB	120 GB
Tasks	**Map**	640	400	701	1040
	Reduce	80	80	80	80
CPU usage	**Map**	Low	High	High	High
	Reduce	Low	High	High	Medium
Block size		256 MB	128 MB	64 MB	128 MB
Combiner		Yes	No	No	No

Table 5.6: Workloads used in the evaluation

for a job are finished. I have verified that using background reduce tasks did not affect the overall job execution time significantly for each of the jobs.

Table 5.5 gives the most important Hadoop configuration options. All other options (except for URLs and storage directories, etc.) were left at their default. Note that the DFS block size specified in Table 5.5 is used for the output of the jobs; the block size of the input data for each workload is given in Table 5.6. For the TeraSort workload, `dfs.replication` for the output is set to 1 by the job itself.

5.6.1 Workloads

In order to evaluate the model, a number of representative data intensive workloads for MapReduce were needed. Jobs that deal with large amounts of data are preferable, particularly if they also have a large amount of intermediate data.

Unfortunately there is no widely accepted benchmark for MapReduce. There

have been some attempts to create standard benchmark suites for MapReduce such as MRBench [KJH+08] and HiBench [HHD+10], but none have gained widespread adoption. Therefore, I have chosen a set of workloads based on commonly used application types.

To evaluate different scenarios for the model, the workloads were chosen such that there is sufficient variety in the amount of CPU and I/O activity for each of the phases from Figure 5.5 and Figure 5.6. One workload is I/O bound in all phases, two are I/O bound in some phases but CPU bound in others, and one is entirely CPU bound. I believe that this is a representative range of workloads for MapReduce, and this covers the most important edge cases for the model.

Table 5.6 provides an overview of the workloads that were used and their properties. TeraSort [Ter] was taken from HiBench [HHD+10] and is the most I/O intensive of the evaluated jobs. This job performs no processing of its own besides a custom trie-based total ordering partitioner, so most of the CPU costs are from the MapReduce framework itself. A 160GB data size was used instead of the full TeraSort workload, which is sufficient to observe significant I/O interference while keeping processing time reasonable.

WordCount is a simple workload often used to explain MapReduce [DG08] and included as a sample with Hadoop. It is representative of many text manipulation and aggregation workloads in Hadoop. Because it uses a combiner for local aggregation its intermediate data size is very small, the reduce tasks are very short, and all processing is entirely CPU bound. The input data was generated using Hadoop's random-text-writer sample.

Parallel FP Growth [LWZ+08] is a frequent pattern mining algorithm for MapReduce, an implementation of which is provided in Mahout [Apac]. However, due to quality issues with this implementation the custom implementation from Chapter 4 is used instead. The input data is a synthetically generated database [AS94] with 800 million transactions, 100,000 unique items and an average transaction length of 10 items. The Parallel FP Growth workload consists of three jobs; the model works was confirmed to work on all three, but only the second job—which runs the actual PFP algorithm—is data intensive. The first job is essentially equal to WordCount, and the third job is very short and processes very little data. Therefore, only the second job is shown here. Although this job has very high CPU usage in its map and reduce phases, the large size of the intermediate data means it still performs a lot of I/O in the other phases of the tasks.

The well-known PageRank [PBMW99, Cas09] algorithm is part of the HiBench benchmark suite, and the implementation was taken from a test case in SmartFrog [Sma]. The input data is a link graph obtained from crawling the web in 2006 [CTK09]. Like Parallel FP Growth, PageRank consists of multiple jobs. It also iterates several times over the data. The UpdateRanks job is the most data

intensive, so only the results of the first iteration of that job are shown here (the model was verified to work on the other jobs as well). This job has high CPU usage in the map and reduce phases, but also very large intermediate data.

5.6.2 Results

To evaluate the model for the workloads, I compare the following values:

Model The execution time of the map or reduce stage as predicted by the I/O interference model described in Section 5.5.

Actual The actual execution time of the map or reduce stage measured by executing the workload with the specified number of parallel tasks on the cluster from Table 5.4.

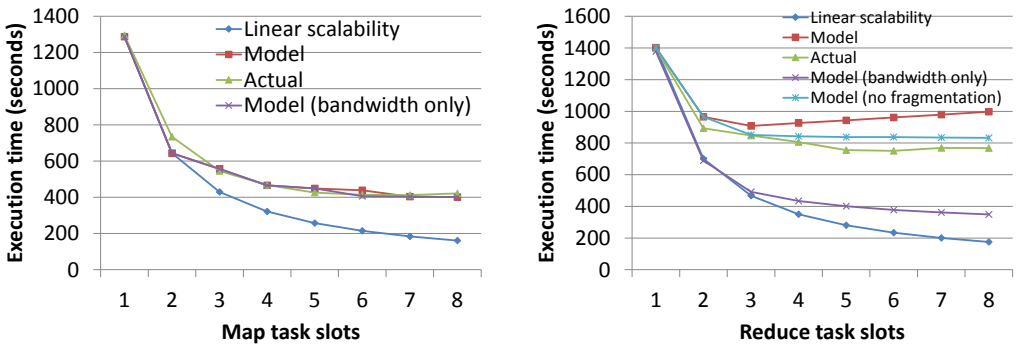
Linear scalability A naive model that assumes that tasks can scale linearly when the number of tasks is increased. It takes the time predicted by the I/O interference model for one slot, and divides it by the number of slots. This is in practice what some of the existing MapReduce performance models (such as [Her10]) do.

Model (bandwidth only) A simple I/O model that accounts for limited bandwidth but not for any other factors. This model takes the time predicted for 1 slot, and then increases I/O costs using Equation (5.3) and (5.4) to account for the bandwidth being split between tasks. Pattern specific costs (for the shuffle phase), hardware specific interference (f_i), and fragmentation costs are not applied, so this comparison shows how much those additional factors impact the result.

Model (no fragmentation) A modification of the I/O interference model that ignores intermediate file fragmentation. Instead of using Equation (5.5) to account for fragmentation, this version of the model simply uses Equation (5.3) for those phases. This is only applied for the reduce stage because no phase in the map stage uses fragmentation.

For each of the workloads two images are shown. The first shows the comparison between the the actual execution and the various estimations listed above. Note that some of the estimation errors may appear inflated because of the rounding that occurs for the shutdown phase; an estimation error of 0.1 seconds can appear as a difference of 3 seconds per task if it happens to cause the rounding to change.

The second figure shows the predicted CPU and I/O cost for each phase in the tasks for the different numbers of parallel tasks. This is not compared to the



(a) Map stage (b) Reduce stage

Figure 5.18: Comparison of model prediction and actual execution time (TeraSort)

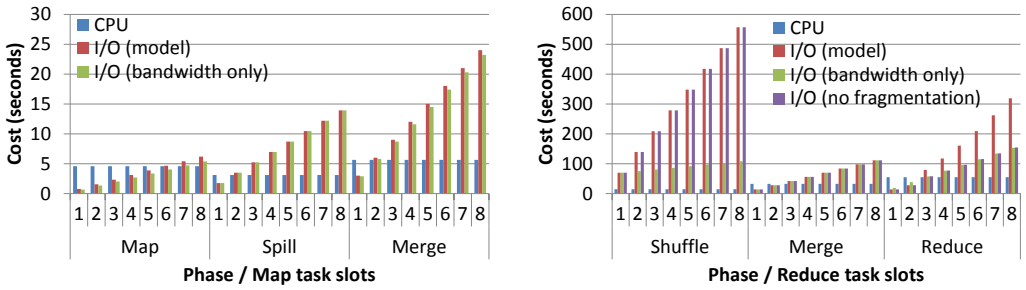
actual values because accurately measuring I/O costs per phase is not possible, as explained in Section 5.4. The I/O costs are shown as predicted by the I/O interference model, the bandwidth only model, and the model without fragmentation. The linear scalability model is omitted from this graph to reduce clutter; the predictions for linear scalability for any number of slots always match the I/O interference model predictions for 1 slot.

For each workload, the model was able to estimate the effect of I/O interference to within 5-10% of the execution time of actual execution, and indicate where the I/O interference occurs even for workloads that were CPU intensive in some parts.

TeraSort

Figure 5.18a shows that there is a difference of 63% between the actual execution time and the one predicted by the naive linear scalability model for map tasks of the TeraSort when running 8 tasks in parallel. For reduce tasks, that difference is even bigger at 77% when running 8 tasks in parallel. This clearly demonstrates the importance of factoring in I/O interference in the prediction. The I/O interference model is able to correctly predict the changes in the execution time to within 5% at 8 parallel tasks for the map stage and 6% at 8 parallel tasks for the reduce stage. Note that it is the prediction without accounting for fragmentation that is more accurate; this is further discussed in Section 5.6.3.

Figure 5.19a shows the effect of interference on the map tasks for TeraSort. The *spill* and *merge* phases become I/O bound when 3 or more tasks are executed in parallel. Although the *read/map/collect* phase remains mostly CPU bound (which is primarily due to the inefficient input format parsing used by this workload; this is an issue with the workload's implementation, not Hadoop), it is clear that at higher number of parallel tasks the *spill* and *merge* phases become significantly



(a) Map stage (b) Reduce stage

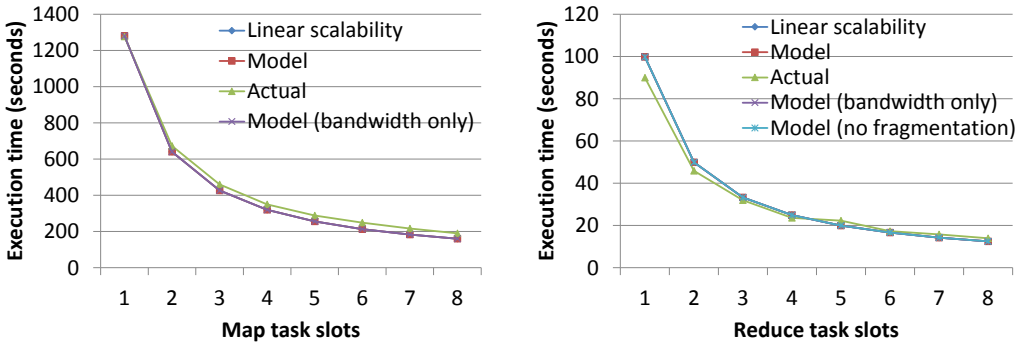
Figure 5.19: Predicted CPU and I/O cost of each phase (TeraSort)

longer than the *read/map/reduce* phase and dominate the task's execution time.

For the reduce tasks, all phases become I/O bound with more than 3 parallel tasks as seen in Figure 5.19b. The shuffle phase in particular starts to dominate execution time at higher numbers of slots, which is a common occurrence for tasks with very large intermediate data sizes in my experience.

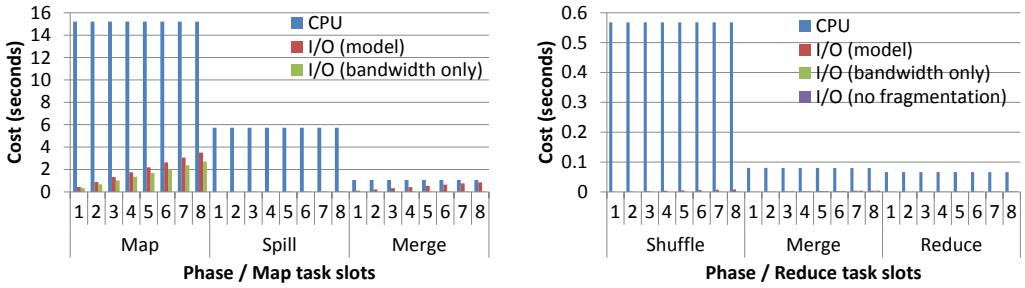
When looking at the bandwidth only model, for the map stage it appears to be identical to the I/O interference model in Figure 5.18a, except at 6 slots where a small difference is show. The only extra interference beyond splitting the bandwidth for map tasks is the hardware specific interference function, f_i, and as shown in Section 5.5.4 that value is not very large for this particular environment. However, on other hardware environments—particularly systems with a single regular hard disk—there can be a considerable amount of interference due to seek costs introduced when the hardware needs to read multiple streams, in which case f_i will be much more complex and there will be a larger difference between the I/O interference model and the bandwidth only model than is the case here. Note that there is actually a small difference between the two models, as seen in Figure 5.19a. However, due to the rounding of task times to account for the shutdown phase, that difference is only visible in Figure 5.18a when it causes a difference in rounding, as is the case for 6 slots.

Figure 5.18b shows that for reduce tasks the bandwidth only model significantly underestimates the execution time, with a 55% error at 8 slots. This is because it does not account for the pattern-specific costs of the *shuffle* phase, which make up the bulk of that phase's I/O cost because of the large number of segments in this workload. This can also be seen in Figure 5.19b, which clearly shows how the bandwidth only approach predicts a much too low cost for the *shuffle* phase. There is also a difference in the I/O cost prediction for the *read/reduce/write* phase, which is due to fragmentation; the bandwidth only model is very close to the prediction without fragmentation in this case because the extra cost of f_i is small for this environment, same as for the map stage.



(a) Map stage (b) Reduce stage

Figure 5.20: Comparison of model prediction and actual execution time (WordCount)



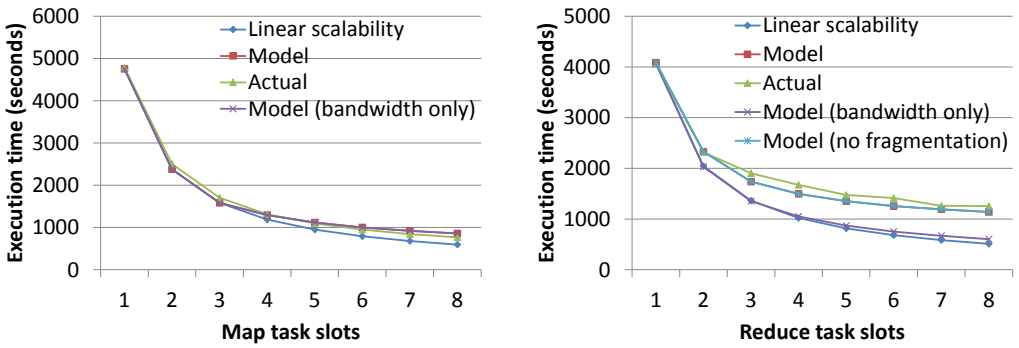
(a) Map stage (b) Reduce stage

Figure 5.21: Predicted CPU and I/O cost of each phase (WordCount)

WordCount

WordCount is used in the experiment to see how the model handles CPU intensive tasks. In the case of a purely CPU bound task that remains CPU bound for all numbers of parallel tasks used in the prediction, the I/O interference model reverts to making the same predictions as the naive linear scalability model. This can be seen in Figure 5.20a and Figure 5.20b, where the lines for *Model*, *Model (bandwidth only)* and *Linear scalability* are identical because I/O interference is not predicted to occur. For the reduce tasks, there is also no difference between *Model* and *Model (no fragmentation)*.

The actual execution time is very close to the model's prediction, indicating that assuming linear scalability is a viable approach for CPU intensive MapReduce workloads. Figure 5.20a shows a small difference between *Actual* and *Model*, which is likely caused by memory or CPU cache contention. However, the effect of this kind of interference is much smaller than the effects of I/O interference observed in



(a) Map stage (b) Reduce stage

Figure 5.22: Comparison of model prediction and actual execution time (Parallel FP Growth)

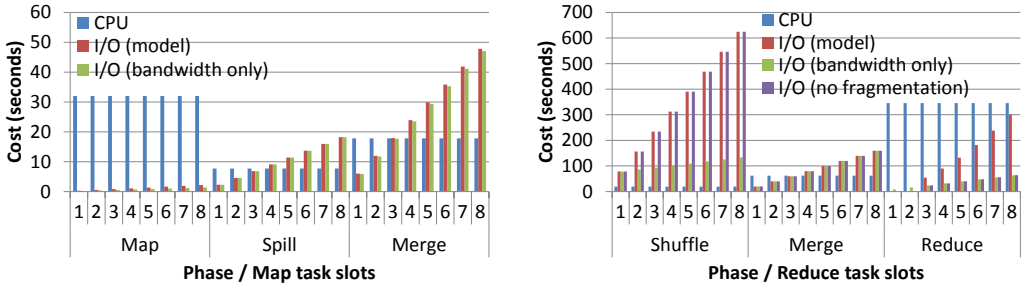
the other workloads in this experiment. Because of its small effect, ignoring these types of interference does not lead to the same kinds of large prediction errors that ignoring I/O interference does.

For the reduce tasks, there is some fluctuation in the value of *Actual*, which is caused by small variations in CPU load due to environmental factors. The reduce tasks of WordCount are extremely short and most of their execution time is framework overhead rather than actual processing, somewhat exaggerating these fluctuations. Nonetheless, *Actual* and *Model* are still very close in all cases.

The CPU intensive nature of the workload can also be observed in Figure 5.21a and Figure 5.21b, where the CPU cost is larger than the predicted I/O cost in all cases. Thanks to the use of a combiner, the intermediate data is very small, and so is the output data. As a result, I/O costs are almost invisible except for the cost of reading the input data in the *read/map/collect* phase. Because of the low amount of I/O, there is also virtually no difference between the different I/O cost estimations.

Parallel FP Growth

Parallel FP Growth has CPU intensive map and reduce functions, but also very large intermediate data so I/O interference is expected to occur. Figure 5.22a shows a 22% percent difference between the I/O interference model prediction and the prediction of the naive linear scalability model when 8 parallel tasks are used. While this is clearly less difference than for TeraSort, it indicates that despite the CPU intensive nature of the map function, I/O interference will stay play a role for the map tasks. The model accurately estimates the actual execution time for the map tasks to within 11% when using 8 parallel tasks, indicating that the model



(a) Map stage (b) Reduce stage

Figure 5.23: Predicted CPU and I/O cost of each phase (Parallel FP Growth)

can handle this type of mixed CPU and I/O workload. As with TeraSort, the bandwidth only model works for the map stage because the contribution of f_i is small on this environment.

The reason for the interference can be seen in Figure 5.23a, which shows that despite the high CPU in the *read/map/collect* phase, I/O interference is still predicted to affect the execution time of the *spill* and *merge* phases when 4 or more parallel tasks are used.

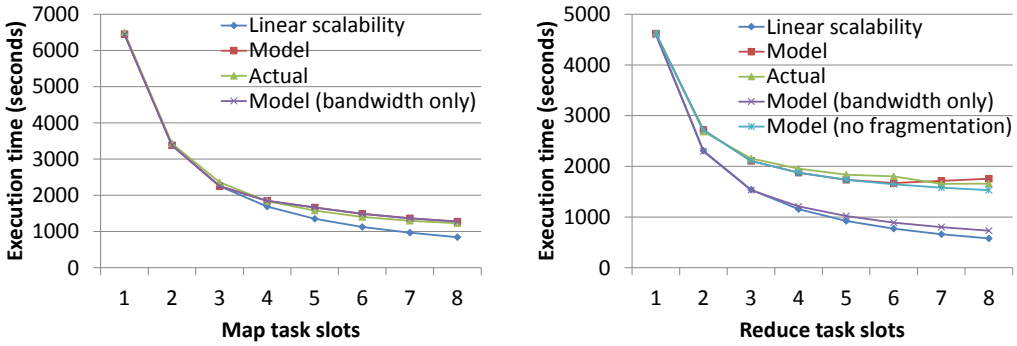
For reduce tasks, Figure 5.22b shows a 59% percent difference between the I/O interference model prediction and the naive linear scalability model at 8 parallel tasks, significantly more than for the map tasks. Once again, the model prediction is very accurate, to within 8% at 8 parallel tasks. The bandwidth only model underestimates the performance by 52% at 8 parallel tasks because it does not correctly account for the *shuffle* phase costs; this is the same phenomenon that was observed for TeraSort.

Figure 5.23b clearly shows why the reduce tasks are more I/O bound, as the *shuffle* phase starts to dominate the execution time due to the large amount of intermediate data despite the high CPU usage in the *read/reduce/write* phase. As with TeraSort, the I/O cost estimation for the shuffle phase made by the bandwidth only model is much too low. Note that for the *read/reduce/write* phase it makes no difference whether intermediate file fragmentation is considered in this case because that phase is CPU bound regardless, primarily because the output data size is rather small.

PageRank

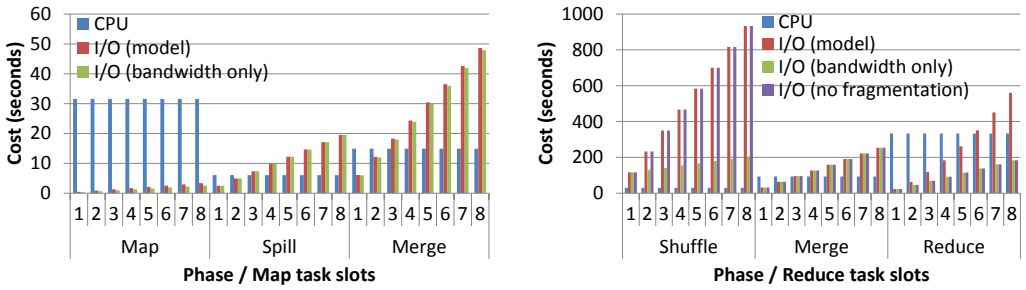
PageRank is similar to Parallel FP Growth because it has a similarly CPU intensive map and reduce function but also very large intermediate data. Unlike Parallel FP Growth, it also has very large output data.

Figure 5.24a shows a difference of 34% between the I/O interference model



(a) Map stage (b) Reduce stage

Figure 5.24: Comparison of model prediction and actual execution time (PageRank)



(a) Map stage (b) Reduce stage

Figure 5.25: Predicted CPU and I/O cost of each phase (PageRank)

prediction and the naive linear scalability model prediction when using 8 parallel tasks. This is a slightly bigger difference than for Parallel FP Growth, because the even larger intermediate data makes the *spill* and *merge* phases more I/O intensive and the *read/map/collect* phase is relatively less CPU intensive, as can be be seen in Figure 5.25a (although the CPU cost in Figure 5.25a and Figure 5.23a seems similar, keep in mind that each map task for PageRank processes twice the amount of data as the Parallel FP Growth map tasks). The model correctly estimates the actual execution to within 5% when running 8 tasks in parallel. For the same reasons as with TeraSort and Parallel FP Growth, there is no significant difference between the I/O interference model and the bandwidth only model.

Figure 5.24b shows that the difference between the I/O interference model and the naive linear scalability model is 64% when using 8 parallel tasks, again slightly bigger than for Parallel FP Growth. The model correctly estimates this value to within 6% at 8 parallel tasks. Figure 5.25b shows that unlike with Parallel FP Growth, the *read/reduce/write* phase is predicted to become I/O bound at

6 or more slots if fragmentation is applied due to the larger size of the output data and slightly lower CPU usage of the reduce function. The bandwidth only model underestimates the costs of the shuffle phase, with a 56% estimation error in Figure 5.24b at 8 slots. This is again for the same reasons as for TeraSort and Parallel FP Growth.

Although there is a difference between the model prediction with and without accounting for fragmentation, it is not significant enough to determine which of the values is correct in this case.

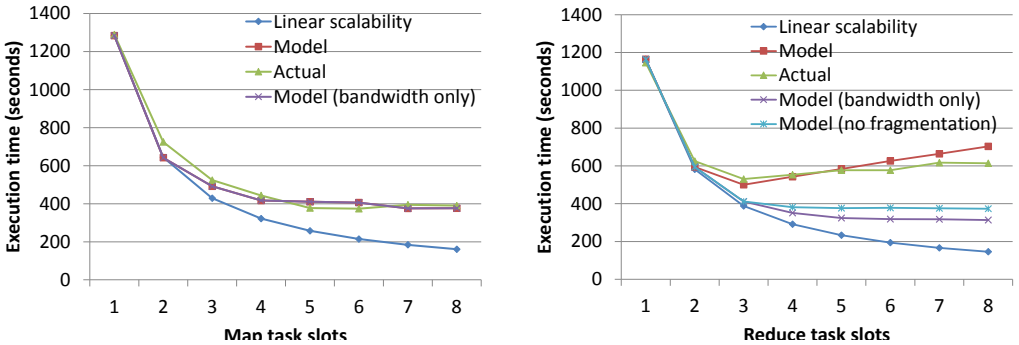
5.6.3 Fragmentation

For TeraSort's reduce tasks there is a significant difference between *Model* and *Model (no fragmentation)* in Figure 5.18b, and *Model (no fragmentation)* is closer to the actual measured execution time. When fragmentation is included, the total execution time for the workload is overestimated by 30% for 8 parallel tasks. Additionally, the model predicts a performance decrease when more than 3 parallel tasks are used which is not observed in the actual execution time.

The prediction of the effect of fragmentation is based on the expected number of parallel writers during those steps in the process when the relevant intermediate files are written, in this case the reduce task *merge* phase. What I observed in this case is that the *shuffle* phase suffered very large variability, with some tasks progressing much faster than others. As a result, not all tasks reached the threshold level for the shuffle memory buffer at the same time, and did not perform background merges at the same time. As a result, the actual number of parallel writers for those intermediate files was fewer than expected.

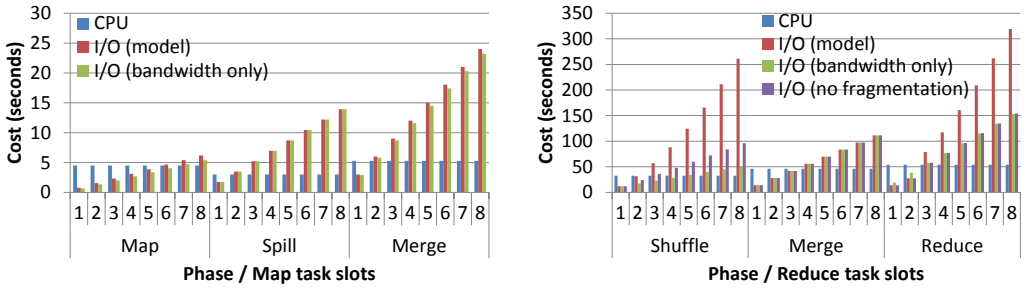
However, this is not always the case. In other circumstances I have observed a significant effect from fragmentation. One such case is when TeraSort is executed on only one node (the data size is decreased in accordance with the number of nodes, as is the number of reduce tasks; on one node, 16 gigabytes of data and 8 reduce tasks were used so the amount of data processed per reduce task is the same as in the experiment with 10 nodes; however, because the number of map and reduce tasks is decreased, the shuffle segment size is larger). When only one node is used, the *shuffle* phase proceeds in a much more predictable fashion thanks to the lower number of segments and their larger size, and most tasks end up performing background merges at the same time.

Figure 5.26 shows the results obtained from running this experiment. In this case the model prediction including fragmentation in Figure 5.26b correctly estimates the performance to within 13% for 8 parallel tasks, while *Model (no fragmentation)* underestimates the execution time by 34%. The actual execution times also demonstrate the predicted decrease in performance when more than 3 parallel tasks are used. While the actual value drops below the prediction for more



(a) Map stage (b) Reduce stage

Figure 5.26: Comparison of model prediction and actual execution time (TeraSort on 1 node)



(a) Map stage (b) Reduce stage

Figure 5.27: Predicted CPU and I/O cost of each phase (TeraSort on 1 node)

than 4 slots because increased task variance begins to affect the fragmentation levels again, the predicted trend is still fairly accurate and the model without fragmentation is clearly incorrect in this case.

Figure 5.27b also shows an interesting thing here: due to the larger segment size, the *shuffle* phase is expected to show fragmentation. This does happen in practice as the prediction in Figure 5.26b would be less accurate if this was ignored and fragmentation was only applied to the *read/reduce/write* phase. That fragmentation is responsible for this difference was further verified by inspecting the actual intermediate files of this job, where I found they indeed showed the expected levels of fragmentation with the accompanying decrease in read performance. The larger segment size is also responsible for the smaller difference between the I/O interference model without fragmentation and the bandwidth only model in this experiment, as there are fewer segments so the additional I/O cost for the shuffle phase is smaller.

Fragmentation can clearly have a very large impact on performance when it

occurs, but it is difficult to know exactly when it will occur or when environmental circumstances cause it to be avoided. One way to address this is to gather more statistics about various workloads and configuration to determine what influences the odds of fragmentation occurring.

5.7 Discussion

5.7.1 Applying model predictions for improved resource utilization

The ultimate goal of this model is to improve resource utilization through improved understanding of I/O interference. This model can help to achieve that by using the information it provides to guide resource provisioning and improve scheduling decisions on several levels such as static scheduling policy, mixed workload scheduling, and dynamic task scheduling.

For all of the I/O intensive workloads in Section 5.6, there are clear diminishing returns on increasing the number of parallel tasks per node. For these workloads, there is usually a point beyond which there is no real benefit anymore.

- For TeraSort map tasks, Figure 5.18 shows that the predicted performance increase when going from 5 parallel tasks to 8 parallel tasks is only 5%, compared to the 67% increase going from 1 to 5 parallel tasks. Therefore, running more than 5 parallel map tasks is not useful. For reduce tasks, the predicted cut-off point is at 3 parallel tasks, as there is only a predicted performance increase of 2% beyond that point (or a 10% performance decrease if intermediate file fragmentation occurs).

- Since WordCount is CPU bound, there is no limit to the number of tasks; linear scalability is predicted in Figure 5.20.

- Although Figure 5.22 predicts some interference for Parallel FP Growth map tasks, the performance is very close to linear. Although the improvements after 6 parallel tasks appear small, they are still 15% going to 8 parallel tasks. Therefore, it could pay off to use 8 parallel tasks if there is no better usage for those CPU cores. The reduce tasks also keep showing improvements because of the CPU intensive nature of the reduce function, although again 6 tasks could be used as the limit with only a 10% improvements from that point.

- With PageRank (Figure 5.24), going from 6 to 7 parallel map tasks is still predicted to show 9% improvement, but from 7 to 8 tasks it is less than 2%. For the reduce tasks, going above 5 or 6 tasks is not very useful as

the performance increase predicted is only 12% or 8% respectively. What's more, if intermediate file fragmentation occurs, performance could decrease by 6% going from 6 to 8 parallel reduce tasks.

The predicted performance improvements (or decreases) for adding additional parallel tasks can be used to guide provisioning decisions. When the pricing of different instance types is taken into account the model can be used to predict the price/performance trade-off between having larger instances running more tasks in parallel, or larger numbers of small instances.

This information can also be used for scheduling a workload. Model predictions could be utilized by the scheduler to limit the number of parallel tasks on a node for a given workload, preferring to use additional nodes rather than additional tasks on the same node when possible, improving overall performance.

This is particularly helpful when multiple workloads or task types are present in the system at once. For example, the scheduler could decide to run 3 TeraSort reduce tasks on a node and fill the remaining capacity of the node with WordCount map tasks or a limited number of Parallel FP Growth map tasks. Although many MapReduce workloads are I/O bound, small CPU intensive workloads do occur relatively frequently; for example, the Parallel FP Growth and PageRank workloads consist of multiple jobs; some of the other jobs in those workloads (not shown here) have highly CPU intensive characteristics not dissimilar to WordCount, so they could be mixed with I/O intensive jobs from other workloads to more efficiently use a node's resources.

Of course, in order to accurately decide how to mix different kinds of tasks from different workloads the model needs to be extended to predict the interference effects of heterogeneous tasks. This is part of the future work for this research.

The model can also be used for more dynamic scheduling decisions. When a new task must be scheduled on a node, information about the currently active tasks and the I/O load of the node will allow an estimation of the performance impact of possible new tasks. This can also benefit from the per-phase estimations provided by the model. For example, Figure 5.22b shows that it is only really the *shuffle* phase that is I/O intensive for Parallel FP Growth reduce tasks. The scheduler can therefore decide to run another I/O intensive task on a node that is running Parallel FP Growth reduce tasks, as long as those tasks have already finished their *shuffle* phase. Predictions at this level are necessarily less accurate due to variability between the tasks, but it is likely that using heuristics derived from the model to guide scheduling in this way can still offer a significant benefit.

Although scheduling in the current version of Hadoop is based on slots and does not take resources into account, the current alpha version of Hadoop (0.23) revises the MapReduce architecture significantly to use a resource-aware scheduling approach, eliminating the notion of slots. This system, called YARN, only supports

memory in the first version, but this model could be used to add I/O support to the YARN scheduler.

5.7.2 I/O interference in other applications

The problem of I/O interference does not just affect MapReduce, but is a problem for all data intensive applications. The model presented in this paper is tailored for MapReduce, but the approach itself can apply to other data intensive applications. My method breaks down a complex workload into smaller pieces (for MapReduce, these were tasks and phases; for other applications, it can be derived for example from database logs), models the I/O behavior of each part separately, and uses an application model to combine the parts and make statements about the whole application. Once the I/O behavior of an application is known, my approach for determining the impact of interference would be applicable to those applications as well.

5.8 Conclusion

In this chapter, I investigated the effects of I/O interference on MapReduce applications. In particular, I focused on the issues that occur when multiple tasks are running in parallel on a single node in order to try and take advantage of multiple CPU cores in those nodes. These tasks need to contend for the same limited storage I/O resources, leading to interference.

In order to enable better understanding and mitigation of I/O interference, I proposed a cost model that is able to predict the effect of I/O interference on the performance of MapReduce workloads when the number of parallel tasks per node is increased. This model uses knowledge about the structure of MapReduce and the I/O patterns it uses to make performance predictions about MapReduce workloads. Interference is handled by hardware specific interference functions, which are derived from micro-benchmarks that measure the performance of the hardware under contention. The combination of the MapReduce and hardware interference models is used to predict how the performance of a workload changes when additional parallel tasks cause interference.

I evaluated the I/O interference model against several representative workloads on a real cluster environment, using a mix of workloads that are either heavily I/O intensive, CPU intensive, or show both CPU and I/O intensive behavior in different parts of the workload. In all cases, the model is able to predict the actual performance of the workload to within 10%, and provides detailed information about which parts of the workload are CPU or I/O intensive, and by how much. It performance much better than a naive linear scalability approach, and also

shows significant benefits compared to only modeling bandwidth, particularly for phases that do not perform sequential I/O (such as the *shuffle* phase) or when fragmentation occurs. Although the hardware specific interference function did not create a large difference on the environment used here, it is likely that it will have a larger impact on different environments.

The information provided by this workload can be used to guide provisioning and scheduling decisions leading to improved resource utilization, which translates into faster execution times and lower costs.

Although the model in this chapter was made for MapReduce, the approach of this model can likely also be applied to other data intensive cloud applications.

Chapter 6

Mariom: MapReduce I/O interference model framework

6.1 Introduction

As covered in Chapter 5, I/O interference affects the performance of many data intensive applications. As multi-core systems become more and more common, interference between applications running on those systems is an increasingly common problem. Therefore, distributed data processing platforms such as MapReduce could benefit greatly from having insights into the I/O behavior of applications to improve scheduling and provisioning decisions.

Provisioning resources for data intensive processing is a complex issue, as there are many factors that can affect the performance of an application. Traditionally the provisioning of compute resources is left up to IT professionals employed by large corporations with sufficient funds to deploy their own infrastructure. These professionals would be experts on distributed systems and are able to make at least an informed decisions based on their previous experience even if they do not know the exact requirements of the workloads they are dealing with. Yet most cloud computing environments move these kinds of decisions to the—often layman—end users. These users have to make decisions about how many nodes of what type to provision. Users are often left merely guessing what configuration is best suited for their workloads and performance requirements, and which options offer the best price/performance trade-off.

Scheduling is the other side of the problem. Once a cluster is provisioned, it is not necessarily straight forward to utilize the provisioned resources efficiently to gain maximum performance from the resources available to the application. The traditional MapReduce scheduler used by Hadoop simply attempts to schedule as many tasks as possible on as many nodes as possible, with the maximum number

of map or reduce tasks per node defined by the number of slots, which is a system wide parameter used for all workloads; the only consideration for efficiency during scheduling is the data locality of map tasks. In the current under development version of Hadoop (Hadoop 0.23), this scheduler will be replaced by YARN, a resource manager that does away with the notion of slots and instead allocates *containers* based on resource requests from the applications. However, in the first version of YARN RAM will be the only supported resource type. Other alternative schedulers for Hadoop typically focus on resource sharing between multiple concurrent workloads, attempting to ensure that one workload cannot starve another of resources or that relative priority levels between the workloads can be enforced.

As the results from Chapter 5 clearly demonstrate, for many applications it is often not a good idea to keep adding additional tasks to a node up to the limit of the memory size or number of CPUs, as this often does not improve performance by much or can even degrade performance due to I/O interference. However, without knowledge about the I/O behavior of applications and the way this behavior changes under interference it is not possible for a scheduler to decide an optimal distribution of tasks among the available resources.

In Chapter 5, I introduced a performance model for MapReduce that is able to predict the effect of I/O interference on an application's performance. In this chapter, I will describe *Mariom*, a framework and toolset for applying this model to MapReduce workloads.

There are already a number of tools for automating Hadoop provisioning, as well as a number of works that attempt to improve Hadoop scheduling. Mariom, in its current incarnation, does not provide either; instead, it focuses on providing a method to utilize the MapReduce I/O interference model in such a way that it could be easily integrated in these kinds of provisioning or scheduling solutions.

6.2 Related work

Improved scheduling for MapReduce is an active area of research, as the performance of MapReduce applications is largely dependent on the effectiveness of the scheduler. Several approaches focus on scheduling in heterogeneous environments [ZKJ+08, TZHZ09, PCB+10, CZG+10, YYH11, FDHG12]. Polo et al. [PCC+11] provide a resource aware approach for dynamically altering the number of slots per node depending on the resource requirements of the applications.

There are not many automatic provisioning tools for MapReduce. One of them is ARIA [VCC11a, VCC11b], which aims to automate allocation of resources to MapReduce applications so that user-specified Service Level Objectives can be met. The Starfish Elastisizer [HDB11] focuses on relative performance models to predict how a workload will perform when moved from one environment to

another, allowing it to optimize the provisioning and Hadoop configuration of the new environment. AROMA [LZ12] similarly enables automatic provisioning based on performance goals, using a two-phase machine learning and optimization framework.

My work does not compete with these existing solutions; rather, it is complementary. Mariom aims to provide I/O interference predictions for a workload and could be integrated into other scheduling or provisioning tools to provide additional information to improve their decision making.

6.3 Design overview

The primary goals when creating Mariom were as follows:

- Provide a method of extracting workload parameters from arbitrary MapReduce workloads, even if they were not executed under ideal circumstances. This makes it possible to deploy Mariom in a production Hadoop cluster and derive workload parameters from ordinary job executions without requiring special workload executions with altered settings.
- Provide the ability to adjust workload parameters based on the workload size and Hadoop configuration. This allows one to derive workload parameters by running only a sample of a complete workload rather than requiring the entire workload to be executed, and also allows the parameters to be used for future executions of that workload with different data sizes and different configuration settings (such as the number of tasks).
- Provide the ability to make performance predictions based on the I/O interference model.
- Implement all parts as reusable Java classes to allow integration of Mariom into more comprehensive scheduling or provisioning solutions, including possibly Hadoop itself.
- Implement the hardware specific parts of the model as plug-ins, so that they can be replaced if Mariom is used on different hardware environments.
- Provide the tools needed to allow derivation of the hardware model for an environment.

Figure 6.1 shows a schematic overview of the design of the Mariom MapReduce I/O interference model framework and toolset. It contains the following components:

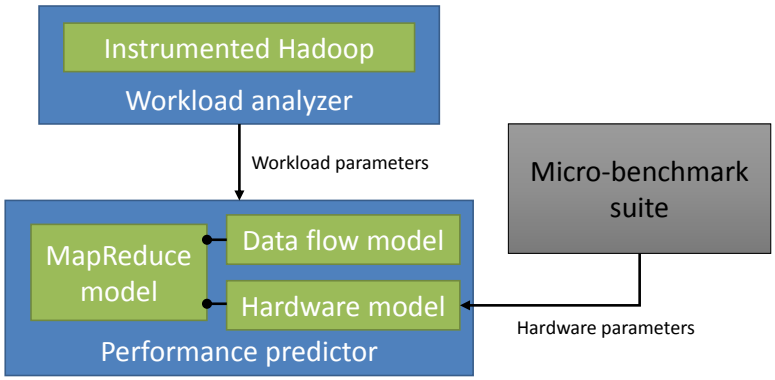


Figure 6.1: Components of the Mariom framework

- The *workload analyzer* derives workload parameters from a given job execution. It includes a modified version of Hadoop that records the necessary performance information to derive the parameters.
- The *performance predictor* makes predictions about a workload given its parameters and the current system configuration. It utilizes a data flow model to adjust the parameters based on changes to the input data and Hadoop configuration, and a hardware interference model to predict the I/O performance under interference.
- The *micro-benchmark suite* provides all the tools necessary to execute the micro-benchmarks from which to derive the hardware parameters and the hardware specific interference function.

Currently, a command line interface is provided to allow for workload analysis and model predictions. This allows a user to manually investigate the performance of a workload under the desired circumstances. As indicated earlier, all components are implemented as Java classes so they can be integrated into full blown provisioning or scheduling systems; the command line interface is simply a convenience provided in the mean time.

The following sections will describe each of the components of Mariom in detail.

6.4 Workload analyzer

The *workload analyzer* is the component of Mariom concerned with deriving the model parameters listed in Table 5.1. In particular, it must derive the data sizes (S_{read} and S_{write}) of each phase as well as the CPU cost (CPU_{phase}). It also collects a number of additional parameters, particularly the number of input and output

records of several of the phases, that are required to support the data flow model described in Section 6.5.1.

Not all of the parameters required for the I/O performance model are normally recorded by Hadoop. It is therefore necessary to hook certain parts of the Hadoop MapReduce framework to support extraction of this information. There are several approaches that could be utilized for this:

- Require workloads to record their own parameters. This is not a desirable solution because it requires alteration of user code for use with the tool. Additionally, user code in the workloads cannot capture statistics about all the parts of the MapReduce framework that are required.
- Use a profiler to derive statistics during execution. This is in many ways the ideal approach, as it requires no modification of existing code. However, while it is easy to profile tasks (this is supported by Hadoop), in practice it turns out some statistics from the Datanode and Tasktracker are also required and it is not so simple to attach a profiler to these processes, especially in a secured environment. Additionally, profilers often have unacceptable levels of overhead that can distort the result.
- Modify Hadoop to record the additional statistics required. While this requires users to run a specialized version of Hadoop, it is the only solution that can capture all the required information and does not have any significant overhead.

Although each solution has its own advantages and disadvantages, I decided to use a modified version of Hadoop as this provides the most accurate information. The modifications to Hadoop are described in Section 6.4.1, and the methods to extract the information is described in Section 6.4.2.

6.4.1 Instrumented Hadoop

Instrumented Hadoop is a modification of the stock Hadoop 0.20.203.0 distribution that records a number of additional parameters about each task and task phase necessary for the model. This is mostly CPU times, but in a few places it was necessary to record additional data sizes and record counts that could not be derived from existing log files or job counters.

All information is recorded in the task log files for later analysis. The exception is the information captured in the Datanode and Tasktracker, which is stored in their own log files.

In order to allow recording of these parameters without requiring special configuration or circumstances during the execution, real CPU times per thread are

recorded rather than elapsed time. Elapsed time is used by some other tools (including Starfish), but this has the problem that it cannot distinguished between CPU and I/O time. Elapsed time is therefore only an accurate measure of CPU time if there is no interference from any other tasks (either CPU or I/O interference) and the workload is entirely CPU bound. In practice, this means that you must utilize only one slot and ensure that Hadoop cannot use more than one CPU (because otherwise multi-threading in some parts (particularly between the task and data server) may lead to under-estimation of the CPU costs. Additionally, it is necessary to ensure that the *read/map/collect* phases do not overlap with the *sort* and *spill* phases (which are executed in a background thread) so that they do not affect each other's elapsed time, and the same is true for the background *merge* phases during the *shuffle* phase. Tools such as Starfish do not actually ensure that this is the case, so depending on the circumstances the measured costs can be highly inaccurate.

Using real CPU times has its own caveats. Measuring the CPU time of a thread has more overhead than simply keeping a timer, and therefore cannot always be done at the desired levels of granularity. For example, the *read*, *map* and *collect* phases are interleaved, so separating their CPU times would require making a measurement for every record read and every record sent to the output. This is prohibitively expensive—it can easily increase the total execution time by an order of magnitude—and the Linux kernel does not keep CPU time statistics at that level of accuracy anyway so the results would not be correct.

While the I/O interference model generally does not require to know each individual CPU time in that level of detail, the data flow model in Section 6.5.1 does for a few cases. In order to handle these, I also capture elapsed time and use the relative times to divide the CPU time. This is not a 100% accurate approach, but it is not possible to improve on this with current operating system limitations.

Another limitation is that some CPU time (such as time spent in the garbage collector and other internal JVM threads) cannot be captured. However, I found this does not affect the accuracy by a significant amount.

Below it is explained for each phase from Figure 5.5 and Figure 5.6 what information is recorded and what changes were made to Hadoop to accommodate this. Most of the changes are relatively small, so it should not be complicated to port these changes to other versions of Hadoop.

The map task *read/map/collect* phases

The only additional information that needs to be recorded for these phases is the CPU time, and the elapsed times of each individual phase as described above.

Hadoop 0.20 supports two different APIs for map and reduce functions, referred to simply as the old and new API. These both use different code paths and different

classes, all of which need to be modified to support both APIs.

In order to record the CPU time across all three phases, the functions responsible for running the old and new version respectively both take a thread CPU measurement before and after map execution. They also record the start and end times in the log, from which the total elapsed time of all three phases can be derived. The input record reader takes timestamps for every record read, providing the elapsed time of the *read* phase. The output collectors for both APIs do the same for each collected record, providing the elapsed time of the *collect* phase. The elapsed time of the *map* phase is the difference between the total elapsed time and the *read* and *collect* times.

The *read* phase reads data from the Hadoop DFS, and some CPU costs are incurred in the Dataserver. Because checksum calculation happens in the client process, the CPU costs on the Datanode are not very high, but they are also not so small that they can be ignored.

In order to capture this information, the Datanode was modified to take CPU time measurements during the handling of a `HDFS_READ` request. Since only one thread is used for each client, and that thread does no other operations during the handling of that client, this is relatively straight forward. These CPU times are recorded in the Datanode's *ClientTrace* log. Because MapReduce tasks use their task ID as the DFS client ID (which is recorded in the *ClientTrace* log), it is possible to determine which log entries correspond to which tasks.

The map task *sort* phase

Two additional pieces of information must be recorded here: the number of records sorted during each spill, and the CPU time of the sorting operation. The number of records is required to calculate the CPU cost per record which is necessary for the data flow model.

Sorting happens on the spill thread, so it is a simple matter to take a thread CPU time measurement before and after the sort, and record that time along with the number of records in the task log file. The *sort* phase does not distinguish between the old and new API so this change only had to be made in one place.

The map task *spill* phase

For the spill phase the number of output records of the spill (which may be different than the number of input records if a combiner is used) must be recorded, the CPU time of the spill operation, and the elapsed time of the combiner. As before, the record numbers and elapsed times are only necessary for the data flow model.

CPU time measurements are added before and after each spill, and the value is recorded in the task log file. To get the elapsed time for the combiner, the

total elapsed time of the spill is recorded, and the input record reader and output collector of the combiner are hooked to record the time spent reading input and writing output; the difference between the total time and the read and write times is the elapsed time for the combiner. The output collector of the combiner also provides the number of output records for the spill. The invocation of the combiner is different depending on whether the old or new API was used, so the input record reader and output collector of both APIs had to be altered.

The map task *merge* phase

Because the *merge* phase can have multiple passes, the number of bytes read and written can be larger than the output data size of the map task. Because the *merge* phase can also invoke the combiner on the final pass, the number of output records and the output size is once again possibly smaller. Because the Hadoop counters do not distinguish spill and merge or the various merge passes, this information must be recorded separately.

For each merge pass, the amount of data processed (which is both read and written) is recorded. For the final merge pass the output size and number of records are separately recorded to account for a possible combiner. As with the *spill* phase, the elapsed time of the combiner is also recorded in a similar fashion (the same modifications are re-used here). CPU time is measured by taking thread time measurements before and after the merge.

The reduce task *shuffle* phase

The shuffle phase uses several different threads: one thread to check which map task outputs are available for download, one thread to schedule the fetches, and one or more threads to download the segments. CPU time measurements are added to each thread, and the values are recorded in the task log file.

The shuffle phase downloads data from the Tasktrackers of other nodes, and some CPU utilization takes place inside the map output server in the Tasktracker. CPU time measurements were added to this server, and those CPU times are recorded in the Tasktracker's *ClientTrace* log. The reduce task ID is also added to this log entry so it is possible to determine which costs belong to which task.

The reduce task *merge* phases

There are three merge phases in a reduce task: the background memory-to-disk merge during shuffling, the background disk-to-disk merge during shuffling, and the memory purge and intermediate merge passes that happen after shuffle is complete and before the reduce function is invoked.

CPU time measurements are added to each of these, with the values recorded in the task log file. In addition, the number of records and data size of each background merge is recorded. For the background memory-to-disk merge, the number of output records is recorded separately because the combiner can be invoked, and the elapsed time for the combiner is also recorded in the same way as done for the map tasks. For the intermediate merge passes after the shuffle phase, the modifications made to the Merger class for the map task *merge* phase already record the data sizes of each pass.

The reduce task *read/reduce/write* phases

These phases are handled very similarly to the map task *read/map/collect* phases. CPU time measurement is done for the three phases combined, and elapsed times are determined by measuring total elapsed time, input reading time and output writing time. Once again it is necessary to hook different classes for the old and new API.

As with reading, writing to the Hadoop DFS incurs CPU cost on the Datanode. Because the final Datanode in the replication chain verifies the client computed checksum, this CPU cost can be significant. The Datanode was modified to record the thread CPU times of `HDFS_WRITE` operations and record it in the Datanode's *ClientTrace* log. Two threads are used for handling write requests; one thread that receives the data (and if necessary, forwards it to the next replica), and one thread that sends acknowledgments back to the client. Although most CPU time is spent in the receiver thread, I still record the CPU time of both. As with reading, MapReduce tasks use their task ID as the DFS client ID, making it possible to determine which operations were done by which tasks.

6.4.2 Analyzer

After a job has been executed using the instrumented version of Hadoop, the workload analyzer can process the collected information to determine the parameters.

As input, the analyzer takes a job ID. It retrieves information (including counter values) about this job from the Jobtracker, and then downloads and parses all the task log files, Datanode log files, and Tasktracker log files relevant tot he job.

Instead of a job ID you can also provide a job history file. This allows analysis of jobs after they have been retired by the Jobtracker, or if the Jobtracker has been restarted since the job was executed. In this case, counter values and other job information is parsed from the history file. It is still required for all relevant log files to be available on the cluster (Hadoop cleans up task log files after some time, usually a few days).

The parameters for each of the map task phases are extracted as follows:

Read/map/collect The number of bytes read is determined from the FileInputFormat *bytes read* counter. The input and output record counts are determined from the *map input records* and *map output records* counters. The output data size is determined from the *map output bytes* counter. The CPU and elapsed times are parsed from the task log file and Datanode log files.

Sort The number of records and CPU time are parsed from the task log file.

Spill The number of output records, bytes written, CPU times and elapsed times are determined by parsing the task log file. The *spilled records* counter cannot be used for this purpose because it includes all records written by the spill and merge phases (including intermediate merge passes).

Merge The number of bytes read, number of bytes written, output size, output records and CPU and elapsed times are parsed from the task log file. The total number of records merged in all passes is determined by subtracting the number of output records from the spill phase from the value of the *spilled records* counter.

For the reduce tasks, it is as follows:

Shuffle The number of bytes read and written is determined from the task log file (this is information already recorded by stock Hadoop). CPU times are determined by parsing the task log file and the Tasktracker log files.

Merge For each type of merge separately, the bytes read, bytes written, output size (may be different because of combiner), merged records, output records, CPU time and elapsed time are parsed from the task log file. The memory-to-disk merge, disk-to-disk merge, and memory purge and intermediate merge passes are kept separately to improve calculations done by the data flow model.

Read/reduce/write The number of input records and output records are determined from the *reduce input records* and *reduce output records* counters. The number of bytes read is parsed from the task log file, and the number of bytes written is determined by the FileOutputFormat *bytes written* counter. The CPU times and elapsed times are parsed from the task log file and Datanode log files.

N_{tasks}^{map}	640
***Init* phase**	
CPU_{phase}	2.113
S_{read}	0
S_{write}	0
***Read/map/collect* phase**	
CPU_{phase}	4.162
S_{read}	268435400
S_{write}	0
***Sort* phase**	
CPU_{phase}	2.377
S_{read}	0
S_{write}	0
***Spill* phase**	
CPU_{phase}	2.883
S_{read}	0
S_{write}	273809388
***Merge* phase**	
CPU_{phase}	5.122
S_{read}	327282240
S_{write}	327282240

(a) Map tasks

N_{tasks}^{reduce}	80
***Init* phase**	
CPU_{phase}	3.446
S_{read}	0
S_{write}	0
***Shuffle* phase**	
CPU_{phase}	12.774
S_{read}	2190436704
S_{write}	0
***Merge* phase**	
CPU_{phase}	30.776
S_{read}	0
S_{write}	2190433041
***Read/reduce/write* phase**	
CPU_{phase}	49.502
S_{read}	2190432882
S_{write}	2147483200

(b) Reduce tasks

Table 6.1: Sample workload parameters for TeraSort

This information is saved in a file, preserving the values for each task. Workload parameters are determined by averaging the values of the tasks. Tasks that deviate too much from the average (determined by using the standard deviation) are ignored to make sure that statistical outliers (such as very small tasks that process the tail of the input files) don't unduly affect the average.

Table 6.1 shows an example of the parameters collected by Mariom for the TeraSort workload from Table 5.6 on the environment from Table 5.4. Only the parameters for the I/O model are shown; the additional parameters collected for the data flow model are omitted.

Parameter	Meaning
`mariom.cache.task.percent`	The per-task data size threshold below which data is considered cached, as a percentage of the node's memory
`mariom.cache.wave.percent`	The per-wave data size threshold below which data is considered cached, as a percentage of the node's memory
`mariom.sync.write.task.percent`	The per-task data size threshold above which writes are considered synchronous, as a percentage of the node's memory
`mariom.sync.write.wave.percent`	The per-wave data size threshold above which writes are considered synchronous, as a percentage of the node's memory
`mariom.shuffle.cache.percent`	The threshold of the total map output size per node below which shuffle data is considered cached, as a percentage of the node's memory
`mariom.cpus.per.node`	The number of CPU cores per node
`mariom.node.memory`	The size of each node's RAM
`mariom.hardware.model`	The name of the class implementing the hardware model
`mariom.data.model`	The name of the class implementing the data flow model

Table 6.2: Configuration settings for Mariom

6.5 Performance predictor

The performance predictor is the core part of Mariom. It takes as input the workload parameters for a job and the current Hadoop configuration, and produces cost predictions for each phase and an overall performance prediction for the tasks and workload as described in Chapter 5.

If the input data size, number of map or reduce tasks, or some Hadoop configuration options (such as the spill buffer size or merge factor) has changed, Mariom first uses a data flow model to adjust the workload parameters for the new settings. Based on the parameters, it then predicts the performance using the method described in Section 5.5 based on the specified number of nodes and parallel map and reduce tasks per node. In order to calculate the effect of I/O interference, it delegates to a hardware model that is specific to the targeted hardware environment.

The Mariom predictor has several configurable parameters which are listed in Table 6.2 which affect the caching heuristics (see Section 5.5.2) and provide some system parameters from Table 5.1 that cannot be determined from regular Hadoop configuration settings or by querying the Jobtracker. It also allows the user to specify the class names of the data flow model and hardware models.

6.5.1 Data flow model

It is desirable to be able to adjust the workload parameters based on changes in the input data, job configuration, or Hadoop configuration. This avoids having to measure the workload again after every change, and allows workload parameters to be determined by measuring only a sample of the workload rather than the entire workload.

In order to allow such estimation, Mariom delegates to a data flow model to adjust the parameters based on their original values from the measurement and the new configuration settings. The data flow model is a plug-in; that is, an interface is provided for the data flow model and any class implementing that interface can be used as the data flow model. The interface has only a single method, `modifyParameters`, which takes as input the original parameters and configuration and the changed configuration, and provides as output the updated parameters. Which class is used is specified using the `mariom.data.model` configuration option.

Mariom provides a default data flow model based on the approach described in [Her10]. Based on the input data size and Hadoop configuration for a job, this data flow model does the following:

- The input data size for each map task is estimated based on the total input data size and the number of map tasks.
- The output data size and number of records of the map function is estimated based on the selectivity of the map function in the original measured workload.
- The number of spills is estimated based on the the size of the map output data and the map output buffer size.
- The output data size and number of records of the spills is estimated based on the selectivity of the combine function (if any).
- The number of merge passes is calculated based on the number of spills and the maximum number of file inputs in each pass (the merge factor).
- The output data size of the merge phase is estimated based on the selectivity of the combine function (if any).
- The input of each reduce task is calculated based on the total map task output size and the number of reduce tasks.
- The number of background memory-to-disk merges during the shuffle phase is calculated based on the number of input segments, the size of each segment,

the size of the reduce input buffer, the merge threshold of the input buffer, and the maximum number of in-memory segments for one merge pass.

- The number of background disk-to-disk merges during the shuffle phase is calculated based on the number of segments stored on disk by either the shuffle or memory merge phases, and the merge factor.
- The amount of segments evicted from memory after the shuffle phase is calculated based on the number of remaining segments in memory and the reduce phase input buffer limit.
- The number of intermediate merge passes before the reduce phase is calculated based on the number of on-disk segments and the merge factor.
- The input data size and number of records of the reduce function is calculated based on the shuffle input data, the number of memory merges and the selectivity of the combine function (memory merges apply the combiner; the other reduce task merges do not).
- The output data size and number of records of the reduce function is calculated based on the selectivity of the reduce function.

The estimated CPU and I/O costs for each phase are calculated by the relative change compared to the original measured workload. However, because the I/O cost calculations of this model cannot account for interference, I only use the cost calculations for the CPU costs; for I/O, only the changes in bytes read and written are calculated and used for the further I/O calculations using the model from Section 5.5.

Data flow model limitations

Although this data flow model is reasonably accurate in most cases, it does have some limitations. These limitations are part of the original model and affect not just Mariom, but also the original application of the model in the Starfish Elastisizer [HDB11]. Mariom is therefore at least as accurate as Starfish in this regard, and to my knowledge no better approximations currently exist. The data flow model is a plug-in so it can easily be replaced in case better approaches are created in the future. Some of the limitations of the current data flow model are discussed below.

In order to handle changes in the selectivity of the map, combine or reduce function, the data flow model must be able to distinguish the CPU costs of the *read*, *map* and *collect* phases, the CPU costs of the combiner as part of the *spill* phase, and the CPU costs of the *read*, *reduce* and *write* phases. Because of the way

these phases interleave, this requires a separate measurement for every record in the input and output. Unfortunately, the Linux kernel does not maintain thread CPU times to that level of accuracy, and even if it did the overhead of doing that many CPU time measurements is significant, often increasing the overall execution time by an order of magnitude rendering the measurements highly inaccurate. I can therefore only measure CPU time for the combined phases, not for the individual component phases.

Starfish takes measurements using elapsed time only which can be done at that level of granularity. However, elapsed time measurements can be inaccurate if there was interference during measurement or if the workload is not CPU bound. Due to this, I found the accuracy of Starfish severely lacking in situations where I/O interference occurs.

To handle these situations, I collect durations as well as CPU time, and split the CPU time based on the relative sizes of the durations of the component phases. The accuracy of that is limited if the original measurement was not taken under ideal conditions without interference, but there is no way to get more accurate data with the current Linux kernel. Because predicting changes in the selectivity requires detailed statistics about the input data and the map, combine and reduce functions, which are often not available in a MapReduce environment, this is relatively rare scenario.

The data flow model also considers selectivity to be constant, which it may not be depending on the functions involved. For example, for the WordCount workload from Section 5.6.1 the output of the combine and reduce functions depends on the number of unique words in the input, not the size of the input. However, without detailed knowledge of the properties of the data and the functions involved, it is not possible to make more accurate predictions regarding selectivity.

For most cost calculations, the data flow model assumes that costs change in a linear fashion with regards to the number of input records. This is reasonable in most cases, but for the map task *sort* phase the costs are $O(n \log n)$, where n is the number of records in each spill. However, since records are sorted first by partition and then by key, a decrease in the number of partitions increases the number of comparisons where the key must be compared rather than just the partition number. This data flow model accommodates this by saying sort costs change according to $O(n \log(\frac{n}{r}))$, where r is the number of partitions, however my experiments have shown this to not be entirely accurate. Since actual costs may also be different depending on the actual keys in the input data, it is likely not possible to get a really precise estimation of the sort costs if the number of records or partitions changed. However, since the *sort* phase is often only a small part of the overall costs of a task, this is not a major problem.

Similarly, the costs of the various *merge* phases in the map and reduce tasks are roughly $O(n \log s)$, where n is the number of records and s the number of

segments being merged. However, merging also includes reading input and output and potentially running a combiner, all of which are actually linear in either the input or output size. Since it is not possible to measure CPU time with sufficient granularity to distinguish the costs of each of these steps, all of these are treated linearly. Note that Starfish also treats merging as $O(n)$, even though it does distinguish those costs. Because the merge CPU costs are often not a large part of the task time and merge phases are often I/O bound anyway (at which point the precise CPU time is no longer relevant), this is not a large problem.

6.5.2 Hardware model

The hardware model is responsible for predicting I/O performance based on the hardware parameters from Table 5.1, including the hardware specific interference function. Because the hardware model is dependent on the target environment, it is not a fixed part of Mariom. Instead, it is a plug-in that can easily be replaced with different models for different environments.

For the hardware model, the interference function f_i is split up into four parts: synchronous and asynchronous read and write interference. This is done to ease implementation for those environments in which some of the interference may overlap with the CPU costs of a phase, and some interference may not (e.g. because it takes place as the stream starts reading).

The hardware model interface specifies the following methods:

getAsyncReadInterference Calculates the read interference portion of f_i that may overlap with the CPU costs of the phase.

getAsyncWriteInterference Calculates the write interference portion of f_i that may overlap with the CPU costs of the phase.

getSyncReadInterference Calculates the read interference portion of f_i that may not overlap with the CPU costs of the phase.

getSyncWriteInterference Calculates the write interference portion of f_i that may not overlap with the CPU costs of the phase.

getRandomIoCost Gets the value of C_{random}, the random I/O cost of each segment in the shuffle phase.

getFragmentationFactor Gets the value of F, the cost correction factor for reading fragmented files.

getShuffleFragmentationFactor Gets the value of F for the shuffle phase. This is treated separately because shuffle input files are created under different

Parameter	Meaning
`mariom.hardware.read.throughput`	The average read throughput of the I/O device ($R_{throughput}$).
`mariom.hardware.write.throughput`	The average write throughput of the I/O device ($W_{throughput}$).
`mariom.hardware.random.io.cost`	The random I/O cost for each shuffle segment (C_{random}).
`mariom.hardware.fragmentation.factor`	The fragmentation factor (F).
`mariom.hardware.shuffle.fragmentation.threshold`	The minimum segment size before fragmentation is applied to the shuffle phase ($F^{shuffle}_{threshold}$).

Table 6.3: Configuration settings for the sample hardware model

conditions, so the hardware model may determine separately whether fragmentation should be applied depending on for example the value of $F^{shuffle}_{threshold}$.

getReadCost Gets the basic read cost of a phase, without interference, based on the number of bytes read and the value of R_{disk}.

getWriteCost Gets the basic write cost of a phase, without interference, based on the number of bytes written and the value of W_{disk}.

The core MapReduce model in the predictor uses these functions to calculate the costs for each of the I/O patterns described in Section 5.5.2.

Since the hardware model is specific for the hardware environment, one must be implemented for each environment after the various parameters and functions have been determined using the micro-benchmarks and the methods described in Section 5.5.4. Once those functions and parameters are determined, it is very straight forward to implement a hardware model class for that environment.

A sample hardware model is provided with Mariom for the hardware environment described in Table 5.4, using the parameters and functions derived for this environment in Section 5.5.4.

The sample hardware model uses several configuration settings for the values of some of the hardware parameters, which are listed in Table 6.3. This makes it possible to utilize the sample data model class as a base class for a custom data model, overriding only the interference functions and using the configuration settings to adjust the value of the other parameters.

6.6 Micro-benchmark suite

The Mariom micro-benchmark suite provides the tools necessary to run the benchmarks that are used to derive the parameters and interference functions of the

hardware model. This collection of micro-benchmarks was used in Section 5.5.4 to derive the hardware model for the environment described in Table 5.4.

While Mariom provides all the micro-benchmarks for deriving the hardware model, it does not automatically derive the actual parameters or functions from the result of the benchmarks. It is up to the user to interpret the result of the benchmarks and derive the necessary information. For some values, such as the basic read and write cost, this is relatively simple. For others, such as the interference function, it is expected to be far more complex depending on the hardware environment. Automatic derivation of hardware performance models is a complex research topic and is outside the scope of this thesis.

The micro-benchmark suite consists of two parts: the I/O benchmark tool that is able to perform the required I/O operations, and a series of scripts that will invoke that tool to perform the necessary benchmarks.

6.6.1 I/O benchmark tool

The I/O benchmark tool is a custom-written C++ application that can perform I/O operations related to the I/O patterns observed in MapReduce as described in Section 5.4. It provides the basic functionality upon which the actual micro-benchmarks for the I/O interference estimation are built.

The functions of the I/O benchmark tool are described below.

Read benchmark

The read benchmark will perform a sequential read of the specified input file or block device. It will open the the specified file, read data from the file into a buffer, and repeat this until end of file or the specified maximum size is reached. The buffer size is specified by the user, and reading can optionally start at an offset into the file. The file will be opened using the `O_RDONLY` flag. Optionally the `O_DIRECT` flag can be used as well to bypass the page cache.

While reading, a timestamp is recorded at specified intervals (intervals are specified in terms of the amount of data read), with at most one timestamp for every buffer read. Optionally, system and process CPU usage can be recorded as well (this is not recommended for small intervals, as the overhead of measuring this often is significant). At the end of the benchmarks all timestamps are written to a file to allow creation of a graph showing throughput over time (similar to Figure 5.12) using gnuplot or similar graphing tools.

The read benchmark can optionally perform a number of random calculations during the read loop to simulate CPU load which may affect I/O scheduling in some environments. The user specifies a *compute factor*, which is the number of

times the random calculation is repeated each iteration, to enable simulation of various CPU and I/O loads.

Write benchmark

The write benchmark performs in much the same way as the read benchmark, except it performs a sequential write. It will create a new file or optionally overwrite an existing one (note that overwriting an existing file has very different performance characteristics than writing a new file and this is not an action Hadoop performs; this option is primarily provided for writing directly to block devices). If an existing file is used, an offset to start writing at can be specified. It can also write directly to a block device, but note that this is destructive to any file system structures that were on the disk and should not be attempted while the disk is mounted. The file will be opened using the `O_WRONLY` and `O_CREAT` flags and optionally also `O_DIRECT` and/or `O_SYNC`.

The benchmark will generate a buffer with random data (the size of which can be specified by the user) and this buffer is written repeatedly to the specified file up to the specified maximum size. Optionally, the buffer can be filled with new random data before every write.

As with the read benchmark, timestamps are collected at specified intervals to allow the creation of a time-line of write throughput. CPU load simulation during writing is also supported.

Read/write benchmark

The read/write benchmark reads from one file and writes the contents to another file. A buffer's worth of data is read from the input, and then immediately written to the output. It is possible to specify a selectivity to write more or less data than was read.

Reading and writing is otherwise done exactly like the read and write benchmarks, including the creation of a throughput time-line and optional CPU load simulation.

Segment read benchmark

The shuffle read benchmark reads from a specified set of files, but only reads up to the specified size from each file before moving on to the next. Reading of each file starts at either a specified or a random offset, and the list of input files can be randomized. This simulates the I/O behavior of the shuffle phase.

The same options as with the read benchmark apply, and as with the other benchmark a throughput time-line is recorded and CPU load can be simulated.

6.6.2 Benchmark scripts

The micro-benchmark scripts utilize the I/O benchmark tool to perform the actual benchmark operations described in Section 5.5.4.

All scripts use the following steps:

1. Creation of the necessary input files. If the benchmark does not need any input files or they already exist, this step can be skipped.
2. Running the pre-benchmark script. This script can be customized by the user to perform additional actions such as starting blktrace, sysstat or other system monitoring tools, or any other steps the user wishes to perform before the benchmark is executed. The default version of this script does the following:
   (a) Clear the operating system page cache by running `sync` and `echo 3 > /proc/sys/vm/drop_caches`. Note that this step requires superuser privileges.
   (b) Read a specified file as "warmup" using the benchmark tool. This has the effect of removing any benchmark-related data from any hardware caches that the previous step could not clear. It also ensures that the I/O device is active and spinning, and that the executable code for the I/O benchmark tool is loaded back into memory after the cache was cleared so this does not cause additional I/O operations during the benchmark.
3. Run the specified benchmark. Depending on the settings, multiple copies of the benchmark tool will be started simultaneously to gather information about multiple parallel readers/writers.
4. Wait for all benchmark invocations to finish.
5. Run the post-benchmark script. By default this script does nothing, but it can be used to stop tools such as blktrace and gather the data they collected.

The individual benchmark scripts are further described below.

Device performance benchmarks

These benchmarks open the specified block device in `O_DIRECT` and read or write it from begin to end (or optionally only partially). This allows the user to chart how device performance changes depending on the offset from the device's first sector. Note that performing the write benchmark will destroy all data on the

disk, including boot sectors and file system structures. Opening a block device directly requires superuser privileges. Optionally, a user-specified number of parallel streams are used.

No input files need to be created for this benchmark. As warmup, the benchmark script reads from the specified device at an offset sufficiently different from the start offset of the actual benchmark.

Sequential read file I/O

This benchmark will first create a number of input files. Because the goal is to establish baseline performance, it is necessary to avoid fragmentation as much as possible. Therefore, ensure that the volume is defragmented and has plenty of contiguous free space available. Input files are created using the `preallocate` system call for file systems that support it.

The benchmark itself will read the files back, using a user-specified number of parallel streams and optionally using `O_DIRECT`.

Sequential write file I/O

This benchmark will write a number of files to the specified device, using a user-specified number of parallel streams. Files will be written using `O_SYNC`, although this can be disabled if desired (in that case, the benchmark itself will mainly measure the speed at which the process can write data into the page cache, but when combined with other tools it can still be useful to investigate write-back behavior of a device).

Shuffle I/O

For input, this benchmark generates a very large (typically at least 256) number of files. The same concerns for avoiding fragmentation as with the regular read benchmark apply. It will then use the benchmark tool's segment read option to read these files back using varying segment sizes and optionally using multiple parallel readers.

Fragmentation

This benchmark works the same as the sequential file read I/O benchmark, but the input files are generated using a varying number of parallel streams to deliberate cause fragmentation by interference. These files are then read back using only a single reader to measure the effect this has on read performance. This is done for different numbers of parallel streams so that the fragmentation factor can be determined.

Shuffle fragmentation

This functions the same way as the the regular shuffle I/O benchmark, only like the fragmentation benchmark the input files are created using varying numbers of parallel streams.

These files are then read back using varying segment sizes to allow the user to determine the value of the threshold segment size for fragmentation in the shuffle phase.

6.7 Experimental evaluation

The functionality of Mariom was evaluated by using it to analyze the workloads described in Section 5.6.1 and predict their performance, using the hardware environment from Table 5.4 and the Hadoop configuration in Table 5.5.

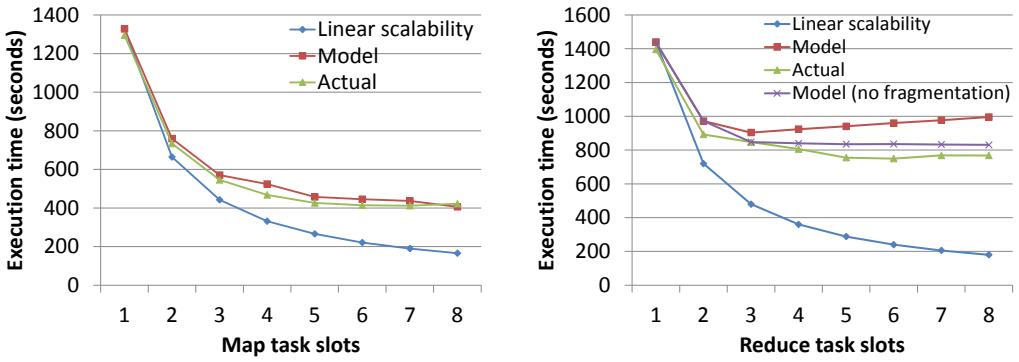
In fact, all the graphs provided in Section 5.6.2 were created using the results from Mariom, so the results shown in Chapter 5 are results for both the model itself and Mariom.

The results demonstrate that Mariom is able to extract the necessary parameters from the workloads and can accurately predict their behavior under I/O interference once the hardware model is established. Since the hardware model was created using the Mariom micro-benchmark suite, this demonstrates their utility in determining the I/O interference behavior of a hardware environment.

It is possible to utilize these results to make scheduling or provisioning decisions. For example, the results for TeraSort in Figure 5.18 show that if you have the choice between provisioning nodes with 4 or 8 cores (and otherwise the same hardware specifications), there is little or no benefit to using the nodes with 8 cores. If the pricing of different node types versus the number of nodes is known, this can be used to recommend what cluster configuration to use to reach a certain performance goal. A scheduler can use this as well; if you can schedule more TeraSort tasks on one node or schedule tasks from a more CPU intensive workload, the latter would be better in terms of maximizing cluster throughput. By contrast, for WordCount shown in Figure 5.20, nearly linear scale-up can be achieved by using the nodes with more CPU cores.

Other information can be included for different trade-offs such as power consumption; disabling half the CPU cores likely won't halve your node's power consumption, while running fewer nodes is more effective in saving power. Information provided by Mariom can help a tool decide how to balance power consumption and performance in such a case.

A scheduler can also integrate the per phase predictions such as those shown in Figure 5.25 for PageRank. That figure shows that PageRank reduce tasks are



(a) Map stage (b) Reduce stage

Figure 6.2: Model predictions for TeraSort using parameters obtained from a job execution with interference

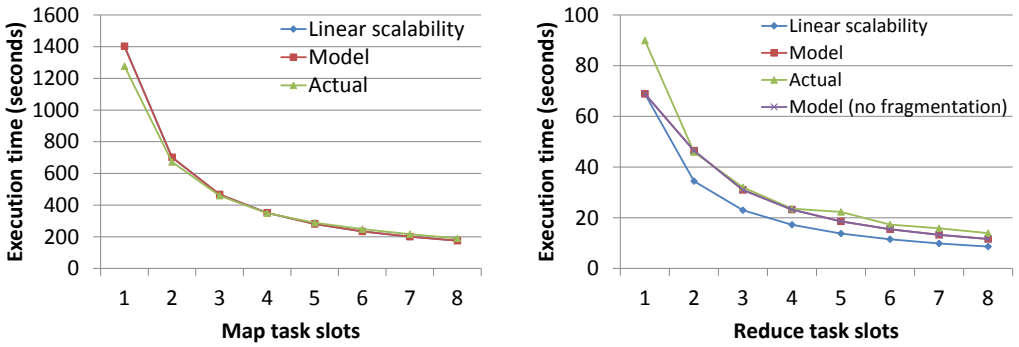
I/O bound primarily during their lengthy shuffle phase, and less so afterwards. Scheduling another I/O intensive tasks on the same node as a PageRank reduce task is therefore better if that task has already finished its shuffle phase.

Mariom's design as a class library allows it to be integrated into a tool such as the Starfish Elastisizer to improve the accuracy of their predictions in the case that I/O interference occurs. For example, one of the functions of Starfish is to recommend what instance type to use for a certain workload on Amazon EC2. Most of the larger instance types have more than one CPU core, but Starfish is not actually able to accurately predict how the workload behavior would change if parallel tasks are used to exploit those CPU cores (it would always assume linear scalability). By including Mariom in Starfish, it would be able to predict the effects of I/O interference once a hardware model for those nodes has been created, allowing it to make a more accurate trade-off between using larger instances or number of nodes.

6.7.1 Measurement under interference

One of the primary advantages of Mariom is its ability to derive workload parameters based on regular executions of the workload. Because Mariom uses CPU time measurements rather than elapsed time, it can get accurate results even if there was I/O interference during the execution of the workload. The results in Chapter 5 were all obtained using parameters derived from job executed using only 1 slot, ensuring no interference occurred during the execution.

Figure 6.2 shows the predictions for TeraSort based on a job execution that was using 4 slots, a situation that has considerable I/O interference extending the duration of the tasks and, as seen in Figure 5.2, causes high levels of variability in



(a) Map stage (b) Reduce stage

Figure 6.3: Model predictions for WordCount using parameters obtained from a job execution with interference

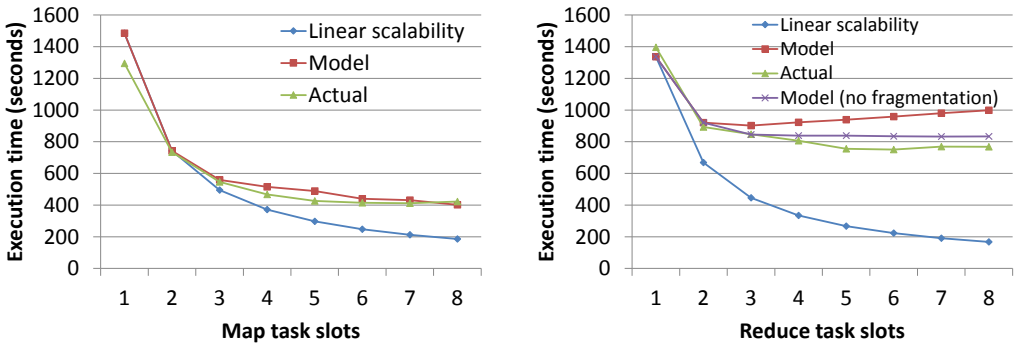
the execution times of the individual tasks.

Despite these conditions, the CPU parameters obtained are still very accurate. For almost all cases the predictions in Figure 6.2 are within 1% of the original predictions from Figure 5.18. The only exception is for the reduce tasks at 1 and 2 slots, where the prediction overestimates the CPU time by 6%. This is because the CPU cost of the *shuffle* and *merge* phases does increase a little due to cache and memory interference, and using these values for lower numbers of slots where these phases are CPU bound causes a slight discrepancy. However, despite this small error the performance trend of the application is still correctly estimated.

Figure 6.3 shows the results for WordCount under the same conditions. For map tasks, shown in Figure 6.3a, the results are very close to the original predictions from Figure 5.20a. CPU costs do increase slightly due to memory and cache interference, so where the original prediction was accurate for low numbers of slot but underestimated the execution time slightly for higher numbers of slots, the situation is now reversed.

For reduce tasks, shown in Figure 6.3b, the prediction is accurate except for 1 slot. However, this is caused by the rounding of the task times for the shutdown phase, which exaggerates what is actually a small difference in he prediction. Because the tasks are very short to begin with, the 3 second difference caused by rounding appears very big although it actually is not. The difference between the model prediction and linear scalability is also caused by this rounding difference, not by actual predicted I/O interference. Nevertheless, the trend of the performance changes is still visible.

Being able to utilize measurements from regular job executions is an important use case. It means that if Mariom is integrated in a production system, it can derive model parameters from any workload execution without having to do special extra



(a) Map stage (b) Reduce stage

Figure 6.4: Model predictions for TeraSort using parameters obtained from a job with smaller input data and fewer tasks

passes to measure the parameters under ideal conditions. If a new, unknown job is executed in such a system it can simply run it normally, analyze the results, and use the parameters from that analysis to optimize future executions of that job.

6.7.2 Data flow model evaluation

Mariom adds a data flow model to be able to utilize model parameters from one execution of a workload for other executions using different data sizes or a different configuration. Although the accuracy of the approach I used by itself has already been evaluated in [HDB11], it is still necessary to determine whether the data flow model yields adjusted parameters that can be used by the I/O interference model for accurate predictions.

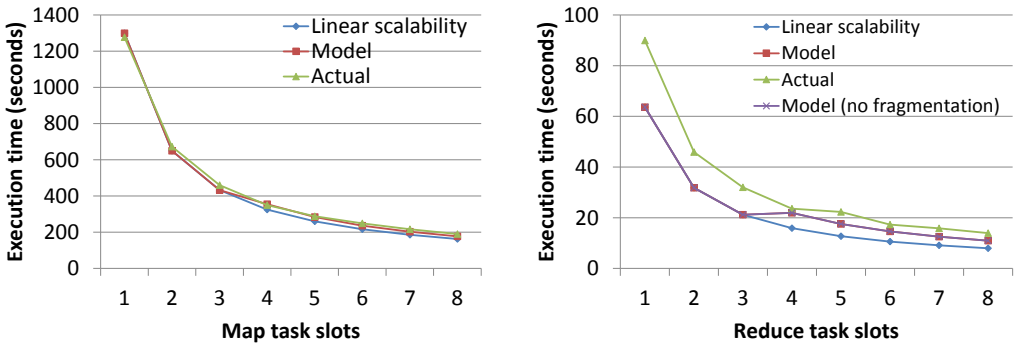
In order to evaluate the data flow model and its interactions with the I/O interference model, I perform two experiments:

1. For the TeraSort and WordCount workloads, I predict the result of the full workload by analyzing an execution using a much smaller data set and corresponding lower number of tasks.

2. For the TeraSort workload, I predict the effect of modifying the number of reduce tasks in tandem with the number of slots.

For TeraSort, I utilize a very small workload with only 2GB of input data with 64 map tasks and 1 reduce task, compared to the 160GB with 640 map and 80 reduce tasks of the full workload.

Figure 6.4 shows the result of predicting the performance of the full workload based on this subset. For the map tasks, shown in Figure 6.4a, the prediction is



(a) Map stage (b) Reduce stage

Figure 6.5: Model predictions for WordCount using parameters obtained from a job with smaller input data and fewer tasks

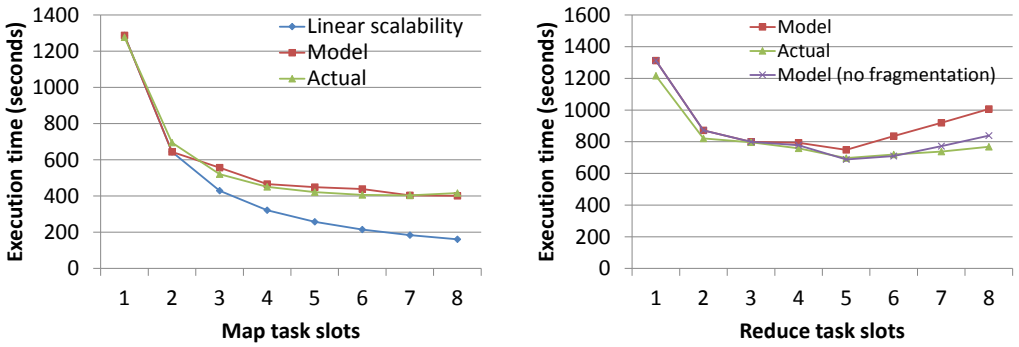
around 5-10% higher than the original prediction in Figure 5.18a, with the biggest difference at 1 slot at 15%. The reason for this difference is an overestimation of the CPU costs of sorting because of the limitations mentioned in Section 6.5.1. Note that the difference is somewhat exaggerated because the task times are rounded based on the shutdown polling interval as explained in Section 5.5.3. For 1 slot, the prediction changes from approximately 20 seconds per map task to 23 seconds per map task, which is the smallest possible difference. In any case, the performance trend is still correctly estimated.

For the reduce tasks in Figure 6.4b, the predictions are mostly within 1% of the original prediction in Figure 5.18b. The higher accuracy is because the longer tasks mean that the differences caused by rounding are relatively smaller, and the fact that most phases are I/O bound at higher numbers of slots means that CPU cost is not so important, and it is easier to accurately calculate the data sizes of each phase than to adjust the CPU costs based on the changes. For 1 slot, the execution time is underestimated by 4% because of an estimation error in the CPU costs of the merge phase again due to the limitations mentioned in Section 6.5.1.

For WordCount, I used a small workload with 1GB input data, 8 map tasks and 1 reduce tasks, compared to 50GB, 400 map tasks and 80 reduce tasks in the full workload.

Figure 6.5 shows the results of using this subset for predicting the full workload. For map tasks in Figure 6.5a there is basically no difference. Only at 4 slots or higher does a small difference occur, which is again mostly caused by rounding. It is a coincidence that this difference brings the prediction closer to the actual execution time.

For the reduce tasks in Figure 6.5b, there is an apparently large difference especially for fewer than 4 slots. This is caused by the fact that the input size of



(a) Map stage (b) Reduce stage

Figure 6.6: Model predictions for TeraSort adjusting the number of reduce tasks based on the number of slots

the reduce tasks is smaller in the full workload than in the small workload, and the data flow model assumes the output size will shrink accordingly based on the selectivity calculation, but in fact it does not because the number of unique words in the input is still the same. The difference this makes is again exaggerated by rounding, which looks very big here because the reduce tasks are extremely short to begin with (only 12 seconds).

Finally, I look at adjusting the number of reduce tasks based on the number of slots for TeraSort. It may be considered desirable in some cases to always have the reduce phase finish in a single wave, and this experiment tests if Mariom is capable of predicting what happens in such a scenario. For each execution, the number of reduce tasks is set to the number of slots per node multiplied by the number of nodes. So on this cluster, for 1 slot there are 10 reduce tasks, for 2 slots there are 20, etc.

Figure 6.6 shows the result of this prediction. There is no line for linear scalability in Figure 6.6b because it is difficult to determine what linear scalability means when the number of tasks is different for every data point.

For the map tasks in Figure 6.6a, the changed number of reducers alters the costs of sorting slightly. However, this has no significant effect on the accuracy of the predictions.

For the reduce tasks, the behavior shown in Figure 6.6b is very different because of the changing number of tasks. Fewer tasks means that the amount of data processed by each task is larger, which changes the behavior of the *shuffle* and particularly the *merge* phases. This is responsible for the somewhat high execution times for 2, 3 and 4 slots. The model is still able to predict each value to within 9% (using the value without fragmentation). Although some overestimation of the execution time occurs at 1 and 2 slots because the data flow model does not

entirely accurately calculate the costs of the merge phase, the model still predicts the trend of the performance with high accuracy, including the fact that 5 slots with 50 reduce tasks is the fastest configuration.

These results show that despite the limitations, the data flow model is capable of predicting at least the trend in the performance changes when the input data size or job configuration is altered. This allows Mariom to use parameters measured from one job execution in different executions of the same job but with different data or settings. It also enables sampling the workload to measure the parameters rather than requiring the user to execute the full workload.

6.8 Conclusion

In this chapter, I have introduced Mariom, a framework and toolset based on the I/O interference model presented in Chapter 5.

Mariom provides the user with three primary components:

- A workload analyzer that can extract workload parameters, including a modified version of Hadoop that measures all the required performance statistics.
- A performance predictor that uses the I/O interference model to predict the behavior of a workload based on the parameters extracted by the analyzer. It uses a data flow model that can adjust parameters based on changes in the data and configuration, and a plug-in hardware model that can be customized for a specific target environment.
- A suite of micro-benchmarks that can be used to derive the hardware parameters and interference functions for an environment.

Mariom is designed as a Java class library to allow it to be integrated into larger toolsets such as provisioning tools or Hadoop schedulers. It enables such tools to create more accurate predictions about the workload performance when I/O interference occurs. Using this information, it is possible to improve provisioning and scheduling decisions.

Due to the high accuracy in predicting performance trends of the model itself, Mariom provides valuable information about workload performance under conditions that previous tools such as the Starfish Elastisizer are not able to handle accurately, or at all. Integrating Mariom into such a tool would allow it to greatly improve its accuracy for these situations.

I have shown that Mariom is able to derive workload parameters even when the workload was executed under less than ideal conditions where interference occurred. This allows it to be utilized in a production system, using real executions

of jobs to determine the parameters rather than requiring a separate pass. With the addition of the data flow model, which I showed to accurately predict the performance trends with changed job settings, these parameters can then be used to predict the performance of future executions of that job even if they use different data sizes or configurations than the original.

This makes Mariom a versatile tool that can be used to provide performance predictions under many real world conditions, covering situations that the performance models used by existing tools cannot handle.

Chapter 7

Conclusions and future work

7.1 Thesis summary

The theme of this thesis is effective resource utilization in the cloud. Cloud computing offers very large scale resources, available for use to anyone on a pay-as-you-go basis. The unprecedented scale of resources in the cloud makes it more difficult than ever to ensure that all resources provisioned for a workload are utilized in an efficient manner throughout the execution of the workload. Often it is found that some resources are idle part of the time because the work is not distributed evenly across resources, or because over-utilization of one resource hinders performance and limits utilization of more abundant resources.

The scalability of cloud applications is directly relevant to this problem. The payment model of the cloud means that a user can provision ten nodes for one hour at exactly the same cost as provisioning one node for ten hours, so a linearly scalable application would mean that a ten times improvement in execution time is possible without any additional cost. Similarly, a user can provision nodes with better hardware such as more CPU cores and memory in the hopes of improving execution time.

Unfortunately, many factors can limit the scalability of a workload and therefore its ability to utilize the resources of both many nodes or faster nodes efficiently. In this thesis, two of the problems that affect the resource utilization of cloud applications were investigated.

Chapter 2 introduces cloud computing and provides a high level overview to how the problem of resource utilization is relevant to these environments. Chapter 3 looks at data processing frameworks for the cloud, with a focus on MapReduce which has become the de facto standard.

In Chapter 4, I consider the problem of workload balancing. When a workload is imbalanced, it means that not all resources are doing the same amount of work,

so some of the resources can be idle while others are still in use. The idle resources are being wasted, and do not contribute to progressing the workload.

Data intensive applications are typically divided into workers, and workload imbalance happens when workers running on different nodes have different execution times. In this situation, some nodes finish running their workers while workers on other nodes have not yet completed. If there are no more workers to schedule in the workload at that time (either because there are no more workers or because remaining workers are waiting on some condition that has not yet been met), the nodes that have finished are idle and cannot help to alleviate the load of the remaining resources being used.

This problem is often seen with map and reduce task workers in MapReduce. Indeed, the issue of stragglers is a well-known problem in MapReduce. Stragglers are small subset of tasks that are substantially longer than the other tasks in the workload, causing the workload's overall execution time to depend on them. Reduce tasks are particularly sensitive to stragglers.

Three main causes are responsible for stragglers: data skew, which means some workers process far more data than others; processing skew, which means some records in the data take more time to process regardless of their size; and performance heterogeneity, which means differences in task execution time are caused by performance differences between the nodes. Such performance differences can be caused by heterogeneous hardware, but also by environmental factors such as background processes that may be active on some nodes or interference between multiple virtual machines on the same node.

I proposed Dynamic Partition Assignment, a method to mitigate the effects of stragglers in the reduce phase. Dynamic Partition Assignment works by dividing the workload into many more partitions than there are tasks, and dynamically reassigning partitions when imbalance is detected. A large percentage of the partitions is assigned to tasks up front, so they can be transferred in bulk at considerably higher efficiency than doing a large number of small transfers. Imbalance is detected by checking for workers that finish their assignmed partitions before the others, at which point it can receive additional partitions to process either from the set of partitions that had not yet been assigned to a task, or by reassigning a partition from a task that is taking a long time to process its partition.

By using lazy imbalance detection depending only on execution time, Dynamic Partition Assignment is agnostic of the cause of the imbalance and can therefore handle data skew, processing skew and performance heterogeneity without any prior knowledge of either the data or the hardware environment. This gives it an advantage over other approaches that use either sampling or runtime statistics gathering that can only really handle data skew.

Dynamic Partition assignment was implemented in Jumbo, a MapReduce-style data processing system I created for experimentation with workload imbalance,

and evaluated for both heavy data skew due to an inefficient partitioning function and for a heterogeneous hardware environment. With data skew, Dynamic Partition Assignment was able to improve the processing time of the affected stage by 50%, almost entirely eliminating the variation between the tasks and therefore eliminating any stragglers. The execution time was brought to within 10% of the estimated optimally balanced time. It was also shown that Dynamic Partition Assignment can offer improved load balancing granularity at lower overhead than simply increasing the number of tasks, which was found to even reduce performance in some cases particularly for Hadoop. For the heterogeneous environment the benefit was smaller because the variation in task execution time was not as large, but Dynamic Partition Assignment was still able to reduce task variance and improve the execution time by 14%, bringing it to within 6% of the estimated optimally balanced time.

In Chapter 5, I investigate another issue of resource utilization, namely the problem of I/O interference. This problem occurs when multiple worker processes need to access the same resource at the same time.

Many resources in the cloud are very abundant. Particularly CPU cores are present in large numbers, as modern server nodes often utilize multi-core CPUs. However, other resources such as disk storage are more constrained. This creates a tension between wanting to increase parallelism to utilize the abundant resources while being limited by the less plentiful resources.

Storage I/O resources are the largest problem in this situation. Storage devices can range from regular single spinning platter hard disks to SSD drives or high-performance storage arrays, but they are usually presented to the system as a single resource that needs to be shared between processes. In addition, storage devices are often the slowest component in a system, and can show complex non-linear performance degradation when interference occurs. As a result, trying to schedule a large number of processes to utilize many CPU cores does not only show limited benefits due to the shared I/O resource, but can in fact reduce performance. Although high-performance solid state devices mitigate some of the impact of this problem, it is still going to occur to some degree every time multiple parallel processes need to share a single resource.

When I/O interference is detected to occur it is often too late to address it. Reassigning work to other machines often involves moving data to new locations, which can increase the problem rather than alleviate it, at least in the short term. The lazy detection utilized by Dynamic Partition Assignment is unsuited for this problem as it only reassigns work after some nodes finish processing rather than when the I/O interference occurs, so some resources may not be efficiently utilized until other resources become available to move work to (which may not happen at all if there is no imbalance). Therefore, a different solution is necessary for I/O interference, one that can predict the problems before they occur so that

provisioning and scheduling decisions can be altered accordingly.

I proposed a cost model for MapReduce that is able to predict the effect of I/O interference when multiple processes running on a node are contending for that node's disk I/O resources. The model uses hardware specific interference functions that are derived from micro-benchmarks used to observe the behavior of the I/O devices under contention, and combines this with an analytical model of MapReduce's behavior to predict the performance of a workload under interference.

The I/O interference model was evaluated using several representative MapReduce workloads on a real cluster environment. The experiment was comprised of a mix of workloads that are heavily I/O intensive, CPU intensive, or exhibit both CPU and I/O intensive behavior. I found that the model is able to predict the actual performance of all the workloads to within 10%, and provides detailed information about which parts of the workload are CPU or I/O intensive, and by how much.

Based on the I/O interference model I developed Mariom, a framework and toolset for applying the interference model to applications, described in Chapter 6. Mariom provides a micro-benchmark suite that allows users to determine hardware parameters and interference functions for their environment, a workload analyzer that allows the collection of workload parameters by executing the workload or a subset thereof, and a performance predictor that uses this information to predict the behavior of an application when I/O interference occurs. In addition, Mariom integrates a data flow model that allows it to adjust workload parameters if the input data or configuration of the job changes.

Because Mariom takes measurements to derive the parameters based on CPU time rather than elapsed time as used by previous tools it is able to derive workload parameters from regular job executions without any restrictions on the configuration used, and with very little impact on accuracy even if there was interference during the measurement. This, combined with the data flow model that allows measurements based on a sample of the workload rather than the entire workload, means that it is very easy to employ Mariom in a production environment to collect parameters about real jobs and predict their performance under different conditions.

Mariom is designed as a Java class library, allowing it to be integrated into full-blown provisioning and scheduling systems, providing them with increased accuracy in reasoning about situations where I/O interference can be a factor. This allows such tools to improve their provisioning and scheduling decisions, which leads to improved resource utilization and ultimately faster execution times and lower costs. Ultimately, Mariom could also be integrated into Hadoop itself, using it to introduce I/O-awareness to the YARN resource scheduler used in future versions of Hadoop.

Dynamic Partition Assignment and the I/O interference model with its asso-

ciated toolset address different parts of the efficient resource utilization puzzle, and both have shown to have real benefits to improving resource utilization for MapReduce. Dynamic Partition Assignment and the I/O interference model were both proposed in the context of MapReduce, but are not specific to MapReduce and can be extended to other data intensive applications.

7.2 Future work

There are a number of research opportunities presented by the work in this paper, for both workload balancing and the topic of I/O interference.

For runtime workload balancing, the primary opportunity is expanding Dynamic Partition Assignment to work in tandem with approaches used in other works which are orthogonal to mine:

- Dynamic Partition Assignment could be extended to handle map tasks in an approach similar to [KBHR12].
- Large keys could be handled by allowing them to be chopped similar to the approach in [RSU12], though this would either introduce a requirement for sampling or requires a method to split a partition after the fact (although [KBHR12] is able to split partitions, it does so by reprocessing the entire partition for repartitioning, which is not desirable).
- Dynamic Partition Assignment could also be applied to approaches like Sailfish [RRS+12], taking advantage of their more efficient shuffle architecture for improved performance. Because Sailfish indexes the intermediate data, it is possible to reassign work without dividing it into a large number of partitions; instead, another worker could be assigned part of the range of another task, using the index to transfer only the necessary data. In addition, because data is not shuffled by the reduce workers but read directly from the I-files stored on the DFS, this approach would eliminate the duplicate data transfers that Dynamic Partition Assignment in its current form is sometimes subject to. Clearly, combining these two approaches is a very natural thing to do and could lead to a very effective, all-purpose skew mitigation system that is able to handle all causes of imbalance rather than just data skew as is currently the case in Sailfish.

For the I/O interference model, there are still many avenues for future research:

- **Mixed workloads**: The model can be extended to handle heterogeneous workloads, predicting the effect of running different types of tasks simultaneously. This would make the model more suitable for determining the most efficient way to mix workloads of different CPU and I/O intensity.

- **Extended tool support**: The I/O interference framework, Mariom, can be integrated into existing provisioning or scheduling system to allow them to make more accurate decisions and weigh the benefits of more nodes vs. more powerful nodes that could run more tasks in parallel, or how to distribute tasks across already provisioned nodes. Alternatively, Mariom itself could be further developed into such a system.

- **Resources shared between nodes**: although the current model focuses on I/O resources shared between tasks running on the same node, there are other types of shared resources. The network is shared between nodes, where activity of certain nodes can affect the performance of other nodes that need to communicate, for example when a central switch limits the performance. Additionally, storage systems such as Amazon S3 are external to the nodes and therefore also shared. Finally, resources may be shared between virtual machines if they are running on the same physical host.

- **Remote resources**: the model currently makes some simplifying assumptions about how I/O operations performed on nodes other than the one that is running the task affect performance. More explicit handling of map tasks that are not data-local or shuffle I/O operations can help to improve predictions, particularly for a scheduler that needs to know the immediate impact of scheduling a certain task on a certain node.

- **Fragmentation**: Section 5.6.3 showed that when intermediate file fragmentation occurs it has a very large impact on the performance of the application, but it is still difficult to predict when this problem occurs. It is possible to handle this situation by gathering more information about various workload types and environmental conditions, and use this to build a statistical model that can more accurately predict when fragmentation will affect a workload.

- **Determining interference functions**: using micro-benchmarks to determine the hardware specific interference functions is currently a mostly manual process that requires some expertise in this area to perform. Refining the selection of benchmarks and using an automated system to derive the models is a very large and interesting research problem that has still not been adequately solved.

Both the workload balancing and I/O interference approaches proposed in this thesis can also be adjusted for and applied to other data intensive application platforms besides MapReduce.

Bibliography

[AAK+11] Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert Greenberg, Ion Stoica, Duke Harlan, and Ed Harris. Scarlett: coping with skewed content popularity in MapReduce clusters. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 287–300, New York, NY, USA, 2011. ACM.

[ABPA+09] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933, August 2009.

[ADR+12] Ganesh Ananthanarayanan, Christopher Douglas, Raghu Ramakrishnan, Sriram Rao, and Ion Stoica. True elasticity in multi-tenant data-intensive compute clusters. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 24:1–24:7, New York, NY, USA, 2012. ACM.

[AFG+10] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[AKG+10] 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 *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI '10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[Amaa] Amazon. Amazon Web Services (AWS). `http://aws.amazon.com/`.

[Amab] Amazon. Elastic Beanstalk. `http://aws.amazon.com/elasticbeanstalk/`.

[Amac] Amazon. Elastic MapReduce. http://aws.amazon.com/elasticmapreduce/.

[Amad] Amazon. Simple Storage Service (S3). http://aws.amazon.com/s3/.

[Ama10] Amazon. Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/, 2010.

[Apaa] Apache. Hadoop Core. http://hadoop.apache.org/core.

[Apab] Apache. HBase. http://hadoop.apache.org/hbase.

[Apac] Apache. Mahout. http://mahout.apache.org/.

[Apad] Apache. Mumak: Map-reduce simulator. https://issues.apache.org/jira/browse/MAPREDUCE-728.

[AS94] R. Agrawal and R. Srikant. Quest synthetic data generator. IBM Almaden Research Center, San Jose, California, http://www.almaden.ibm.com/cs/quest/syndata.html, 1994.

[BCKR11] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards predictable datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 conference*, SIGCOMM '11, pages 242–253, New York, NY, USA, 2011. ACM.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Operating Systems Review*, 37(5):164–177, October 2003.

[BEG+11] K. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Eltabakh, C.C. Kanne, F. Ozcan, and E.J. Shekita. Jaql: A scripting language for large scale semistructured data analysis. *Proceedings of the VLDB Endowment*, 4(12):1272–1283, August 2011.

[BEH+10] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/PACTs: a programming model and execution framework for web-scale analytical processing. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 119–130, New York, NY, USA, 2010. ACM.

[BHBE10] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. HaLoop: efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, September 2010.

[Bru11] G. Brumfiel. Down the petabyte highway. *Nature*, 469(20):282–283, 2011.

[Bur06] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[Cas09] P. Castagna. Having fun with PageRank and MapReduce. Hadoop User Group UK talk, `http://static.last.fm/johan/huguk-20090414/paolo_castagna-pagerank.pdf`, 2009.

[CCG+11] Yu Cao, Chun Chen, Fei Guo, Dawei Jiang, Yuting Lin, Beng Chin Ooi, Hoang Tam Vo, Sai Wu, and Quanqing Xu. ES2: A cloud data storage system for supporting both OLTP and OLAP. In *Proceedings of the IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 291–302, Washington, DC, USA, April 2011. IEEE Computer Society.

[CDG+08] 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 Transactions on Computer Systems*, 26(2):4:1–4:26, June 2008.

[CGK10] Yanpei Chen, Archana Ganapathi, and Randy H. Katz. To compress or not to compress - compute vs. io tradeoffs for mapreduce energy efficiency. In *Proceedings of the first ACM SIGCOMM workshop on Green networking*, Green Networking '10, pages 23–28, New York, NY, USA, 2010. ACM.

[CH11] R.C. Chiang and H.H. Huang. TRACON: Interference-aware scheduling for data-intensive applications in virtualized environments. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 1 –12, New York, NY, USA, November 2011. ACM.

[Che10] Songting Chen. Cheetah: a high performance, custom data warehouse on top of MapReduce. *Proceedings of the VLDB Endowment*, 3(1-2):1459–1468, September 2010.

[CJkL+08] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, August 2008.

[CTK09] Young-joo Chung, Masashi Toyoda, and Masaru Kitsuregawa. A study of link farm distribution and evolution using a time series of web snapshots. In *Proceedings of the 5th International Workshop on Adversarial Information Retrieval on the Web*, AIRWeb '09, pages 9–16, New York, NY, USA, 2009. ACM.

[CZB11] Lu Cheng, Qi Zhang, and Raouf Boutaba. Mitigating the negative impact of preemption on heterogeneous MapReduce workloads. In *Proceedings of the 7th International Conference on Network and Services Management*, CNSM '11, pages 189–197, Laxenburg, Austria, Austria, 2011. International Federation for Information Processing.

[CZG+10] Quan Chen, Daqiang Zhang, Minyi Guo, Qianni Deng, and Song Guo. SAMR: A self-adaptive MapReduce scheduling algorithm in heterogeneous environment. In *Proceedings of the 10th International Conference on Computer and Information Technology (CIT)*, CIT '10, pages 2736 –2743, jul 2010.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th symposium on Operating Systems Design & Implementation*, OSDI '04, pages 137–150, Berkeley, CA, USA, 2004. USENIX.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[DG10] Jeffrey Dean and Sanjay Ghemawat. MapReduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, January 2010.

[DNS+92] D.J. DeWitt, J.F. Naughton, D.A. Schneider, S. Seshadri, et al. Practical skew handling in parallel joins. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, pages 27–40. VLDB Endowment, 1992.

[DQRJ+10] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: making a yellow

elephant run like a cheetah (without it even noticing). *Proceedings of the VLDB Endowment*, 3(1-2):515–529, September 2010.

[ELZ+10] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 810–818, New York, NY, USA, 2010. ACM.

[FBK+12] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 99–112, New York, NY, USA, 2012. ACM.

[FDHG12] Zacharia Fadika, Elif Dede, Jessica Hartog, and Madhusudhan Govindaraju. MARLA: MapReduce for heterogeneous clusters. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, CCGRID '12, pages 49–56, Washington, DC, USA, 2012. IEEE Computer Society.

[FJV+12] Benjamin Farley, Ari Juels, Venkatanathan Varadarajan, Thomas Ristenpart, Kevin D. Bowers, and Michael M. Swift. More for your money: exploiting performance heterogeneity in public clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 20:1–20:14, New York, NY, USA, 2012. ACM.

[FPST11] Avrilia Floratou, Jignesh M. Patel, Eugene J. Shekita, and Sandeep Tata. Column-oriented storage techniques for MapReduce. *Proceedings of the VLDB Endowment*, 4(7):419–429, April 2011.

[GAW09] Ajay Gulati, Irfan Ahmad, and Carl A. Waldspurger. Parda: proportional allocation of resources for distributed storage access. In *Proccedings of the 7th conference on File and storage technologies*, FAST '09, pages 85–98, Berkeley, CA, USA, 2009. USENIX Association.

[GCF+10] A. Ganapathi, Yanpei Chen, A. Fox, R. Katz, and D. Patterson. Statistics-driven workload modeling for the cloud. In *Proceedings of the 26th International Conference on Data Engineering Workshops*, ICDEW '10, pages 87–92, march 2010.

[GF12] Zhenhua Guo and Geoffrey Fox. Improving mapreduce performance in heterogeneous network environments and resource utilization. In

Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID '12, pages 714–716, Washington, DC, USA, 2012. IEEE Computer Society.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[GKAK10] A. Gulati, C. Kumar, I. Ahmad, and K. Kumar. Basil: automated io load balancing across storage devices. In *Proceedings of the 8th USENIX conference on File and Storage Technologies*, FAST '10, pages 169–182, Berkeley, CA, USA, 2010. USENIX Association.

[GLW+10] Chuanxiong Guo, Guohan Lu, Helen J. Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. SecondNet: a data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International Conference on emerging Networking EXperiments and Technologies*, Co-NEXT '10, pages 15:1–15:12, New York, NY, USA, 2010. ACM.

[GNC+09] Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a high-level dataflow system on top of map-reduce: the pig experience. *Proceedings of the VLDB Endowment*, 2(2):1414–1425, August 2009.

[Gooa] Google. Google App Engine. `http://appengine.google.com`.

[Goob] Google. Google Apps. `http://www.google.com/apps`.

[GR11] J. Gantz and D. Reinsel. Extracting value from chaos. `http://idcdocserv.com/1142`, June 2011.

[GSA+11] Ajay Gulati, Ganesha Shanmuganathan, Irfan Ahmad, Carl Waldspurger, and Mustafa Uysal. Pesto: online storage performance management in virtualized datacenters. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SoCC '11, pages 19:1–19:14, New York, NY, USA, 2011. ACM.

[GTOK02] Kazuo Goda, Takayuki Tamura, Masato Oguchi, and Masaru Kitsuregawa. Run-time load balancing system on SAN-connected PC cluster for dynamic injection of CPU and disk resource – a case

study of data mining application –. In *Proceedings of the 13th International Conference on Database and Expert Systems Applications*, DEXA '02, pages 182–192, London, UK, UK, 2002. Springer-Verlag.

[HBK05] M. Hadjieleftheriou, J.W. Byers, and J. Kollios. Robust sketching and aggregation of distributed data streams. Technical report, Boston University Computer Science Department, 2005.

[HDB11] Herodotos Herodotou, Fei Dong, and Shivnath Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SoCC '11, pages 18:1–18:14, New York, NY, USA, 2011. ACM.

[Her10] Herodotos Herodotou. Hadoop performance models. Technical report, Duke University, 2010.

[HHD+10] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *Proceedings of IEEE 26th International Conference on Data Engineering Workshops*, volume 0, pages 41–51, Los Alamitos, CA, USA, March 2010. IEEE Computer Society.

[HKJR10] P. Hunt, M. Konar, F.P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference*, USENIXATC '10, Berkeley, CA, USA, June 2010. USENIX Association.

[HLZ+11] Yin Huai, Rubao Lee, Simon Zhang, Cathy H. Xia, and Xiaodong Zhang. DOT: a matrix model for analyzing, optimizing and deploying software for big data analytics in distributed systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SoCC '11, pages 4:1–4:14, New York, NY, USA, 2011. ACM.

[HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. *SIGMOD Record*, 29(2):1–12, May 2000.

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Operating Systems Review*, 41(3):59–72, June 2007.

[IZ10] Ming-Yee Iu and Willy Zwaenepoel. HadoopToSQL: a MapReduce query optimizer. In *Proceedings of the 5th European conference on*

Computer systems, EuroSys '10, pages 251–264, New York, NY, USA, 2010. ACM.

[JBC+12] Virajith Jalaparti, Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Bridging the tenant-provider gap in cloud services. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 10:1–10:14, New York, NY, USA, 2012. ACM.

[JCR11] Eaman Jahani, Michael J. Cafarella, and Christopher Ré. Automatic optimization for MapReduce programs. *Proceedings of the VLDB Endowment*, 4(6):385–396, March 2011.

[JOSW10] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The performance of MapReduce: an in-depth study. *Proceedings of the VL*, 3(1-2):472–483, September 2010.

[JQRD11] Alekh Jindal, Jorge-Arnulfo Quiané-Ruiz, and Jens Dittrich. Trojan data layouts: right shoes for a running elephant. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SoCC '11, pages 21:1–21:14, New York, NY, USA, 2011. ACM.

[KBHR10] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 75–86, New York, NY, USA, 2010. ACM.

[KBHR12] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. SkewTune: mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 25–36, New York, NY, USA, 2012. ACM.

[KCSR04] R. Khare, D. Cutting, K. Sitaker, and A. Rifkin. Nutch: A flexible and scalable open-source web search engine. Technical report, Oregon State University, 2004.

[KJH+08] Kiyoung Kim, Kyungho Jeon, Hyuck Han, Shin-gyu Kim, Hyungsoo Jung, and Heon Y. Yeom. MRBench: A benchmark for MapReduce framework. In *Proceedings of the 2008 14th IEEE International Conference on Parallel and Distributed Systems*, ICPADS '08, pages 11–18, Washington, DC, USA, 2008. IEEE Computer Society.

[KKL+07] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, June 2007.

[Kos] Kosmos. Kosmos distributed filesystem. `http://code.google.com/p/kosmosfs/`.

[KPP09] Karthik Kambatla, Abhinav Pathak, and Himabindu Pucha. Towards optimizing Hadoop provisioning in the cloud. In *Proceedings of the First Workshop on Hot Topics in Cloud Computing*, HotCloud '09, Berkeley, CA, USA, 2009. USENIX Association.

[LWZ+08] Haoyuan Li, Yi Wang, Dong Zhang, Ming Zhang, and Edward Y. Chang. PFP: Parallel FP-growth for query recommendation. In *Proceedings of the 2008 ACM conference on Recommender Systems*, RecSys '08, pages 107–114, New York, NY, USA, 2008. ACM.

[LYKZ10] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. Cloudcmp: comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, IMC '10, pages 1–14, New York, NY, USA, 2010. ACM.

[LZ12] Palden Lama and Xiaobo Zhou. AROMA: automated resource allocation and configuration of mapreduce environment in the cloud. In *Proceedings of the 9th international conference on Autonomic computing*, ICAC '12, pages 63–72, New York, NY, USA, 2012. ACM.

[MBG10] Kristi Morton, Magdalena Balazinska, and Dan Grossman. Paratimer: a progress indicator for mapreduce dags. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 507–518, New York, NY, USA, 2010. ACM.

[MBLL07] JamesP. McDermott, G.Jogesh Babu, JohnC. Liechty, and DennisK.J. Lin. Data skeletons: simultaneous estimation of multiple quantiles for massive streaming datasets with applications to density estimation. *Statistics and Computing*, 17(4):311–321, December 2007.

[MG11] Peter M. Mell and Timothy Grance. Sp 800-145. the nist definition of cloud computing. Technical report, National Institute of Standards & Technology, Gaithersburg, MD, United States, 2011.

[Mica] Microsoft. Microsoft Office 365. `http://www.microsoft.com/office365`.

[Micb] Microsoft. Windows Azure HDInsight. `http://www.hadooponazure.com/`.

[Micc] Microsoft. Windows Azure Virtual Machines. `http://www.windowsazure.com`.

[MRL99] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. Random sampling techniques for space efficient online computation of order statistics of large datasets. *SIGMOD Record*, 28(2):251–262, June 1999.

[MWS+07] Michael P. Mesnier, Matthew Wachs, Raja R. Sambasivan, Alice X. Zheng, and Gregory R. Ganger. Modeling the relative fitness of storage. In *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '07, pages 37–48, New York, NY, USA, 2007. ACM.

[ORS+08] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[PAL12] Nohhyun Park, Irfan Ahmad, and David J. Lilja. Romano: autonomous storage management using performance prediction in multi-tenant datacenters. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 21:1–21:14, New York, NY, USA, 2012. ACM.

[PBMW99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[PCB+10] J. Polo, D. Carrera, Y. Becerra, V. Beltran, J. Torres, and E. Ayguad´e. Performance management of accelerated MapReduce workloads in heterogeneous clusters. In *Proceedings of the 39th International Conference on Parallel Processing (ICPP)*, ICPP '10, pages 653 –662, sept. 2010.

[PCC+11] 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 *Proceedings of the 12th International Middleware Conference*, Middleware '11, pages 180–199, Laxenburg, Austria, Austria, 2011. International Federation for Information Processing.

[PI+96] V. Poosala, Y. Ioannidis, et al. Estimation of query-result distribution and its application in parallel-join load balancing. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, VLDB '96, pages 448–459. VLDB End, 1996.

[PLM+10] Xing Pu, Ling Liu, Yiduo Mei, S. Sivathanu, Younggyun Koh, and C. Pu. Understanding performance interference of i/o workload in virtualized cloud environments. In *Proceedings of the IEEE 3rd International Conference on Cloud Computing*, CLOUD '10, pages 51–58, Washington, DC, USA, July 2010. IEEE Com.

[PPR+09] 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.

[Rac] Rackspace. The Rackspace open cloud. `http://www.rackspace.com/`.

[RBG12] Anshul Rai, Ranjita Bhagwan, and Saikat Guha. Generalized resource allocation for the cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 15:1–15:12, New York, NY, USA, 2012. ACM.

[RLC+12] Alexander Rasmussen, Vinh The Lam, Michael Conley, George Porter, Rishi Kapoor, and Amin Vahdat. Themis: an i/o-efficient mapreduce. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 13:1–13:14, New York, NY, USA, 2012. ACM.

[RLTB10] Martin Randles, David Lamb, and A. Taleb-Bendiab. A comparative study into distributed load balancing algorithms for cloud computing. In *Proceedings of the IEEE 24th International Conference on Advanced Information Networking and Applications Workshops*,

WAINA '10, pages 551–556, Washington, DC, USA, 2010. IEEE Computer Society.

[RRS+12] Sriram Rao, Raghu Ramakrishnan, Adam Silberstein, Mike Ovsiannikov, and Damian Reeves. Sailfish: a framework for large scale data processing. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 4:1–4:14, New York, NY, USA, 2012. ACM.

[RSU12] Smriti R. Ramakrishnan, Garret Swart, and Aleksey Urmanov. Balancing reducer skew in mapreduce workloads using progressive sampling. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 16:1–16:14, New York, NY, USA, 2012. ACM.

[RTG+12] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 7:1–7:13, New York, NY, USA, 2012. ACM.

[SAD+10] M. Stonebraker, D. Abadi, D.J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. Mapreduce and parallel dbmss: friends or foes. *Communications of the ACM*, 53(1):64–71, January 2010.

[SAS08] Hongzhang Shan, Katie Antypas, and John Shalf. Characterizing and predicting the i/o performance of hpc applications using a parameterized synthetic benchmark. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 42:1–42:12, Piscataway, NJ, USA, 2008. IEEE Press.

[SDQR10] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proceedings of the VLDB Endowment*, 3(1-2):460–471, September 2010.

[SHB04] Mehul A. Shah, Joseph M. Hellerstein, and Eric Brewer. Highly available, fault-tolerant, parallel dataflows. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 827–838, New York, NY, USA, 2004. ACM.

[Sma] SmartFrog. Smartfrog project homepage. `http://www.smartfrog.org`.

[SN95] Ambuj Shatdal and Jeffrey F. Naughton. Adaptive parallel aggregation algorithms. *SIGMOD Record*, 24(2):104–114, May 1995.

[SSGW11] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. CloudScale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 5:1–5:14, New York, NY, USA, 2011. ACM.

[Swe06] D.W. Sweeney. Overview of the large synoptic survey telescope project. In *Proceedings of the international society for optics and photonics*, volume 6267, pages 1–9. SPIE, June 2006.

[TC11] Fengguang Tian and Keke Chen. Towards optimal resource provisioning for running MapReduce programs in public clouds. In *Proceedings of the International Conference on Cloud Computing*, CLOUD '11, pages 155–162, july 2011.

[Ter] TeraSort. TeraSort. `http://sortbenchmark.org/`.

[TK99] Masahisa Tamura and Masaru Kitsuregawa. Dynamic load balancing for parallel association rule mining on heterogenous pc cluster systems. In *VLDB '99 Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 162–173, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[TSJ+09] A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, February 2009.

[TZHZ09] Chao Tian, Haojie Zhou, Yongqiang He, and Li Zha. A dynamic mapreduce scheduler for heterogeneous workloads. In *Proceedings of the 8th International Conference on Grid and Cooperative Computing (GCC)*, GCC '09, pages 218 –224, aug. 2009.

[VBBE12] Rares Vernica, Andrey Balmin, Kevin S. Beyer, and Vuk Ercegovac. Adaptive MapReduce using situation-aware mappers. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 420–431, New York, NY, USA, 2012. ACM.

[VCC11a] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. ARIA: automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM international conference on*

Autonomic computing, ICAC '11, pages 235–244, New York, NY, USA, 2011. ACM.

[VCC11b] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. Resource provisioning framework for MapReduce jobs with performance goals. In *Proceedings of the 12th International Middleware Conference*, Middleware '11, pages 160–179, Laxenburg, Austria, Austria, 2011. International Federation for Information Processing.

[WBPG09] Guanying Wang, A.R. Butt, P. Pandey, and K. Gupta. A simulation approach to evaluating design decisions in MapReduce setups. In *Proceedings of the International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems*, MASCOTS '09, pages 1–11, sept. 2009.

[WBPR12] Alexander Wieder, Pramod Bhatotia, Ansley Post, and Rodrigo Rodrigues. Orchestrating the deployment of computations in the cloud with conductor. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 27–27, Berkeley, CA, USA, 2012. USENIX Association.

[WDJ91] C.B. Walton, A.G. Dale, and R.M. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases"*, VLDB '91. VLDB Endow, 1991.

[WK09] Daniel Warneke and Odej Kao. Nephele: efficient parallel data processing in the cloud. In *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*, MTAGS '09, pages 8:1–8:10, New York, NY, USA, 2009. ACM.

[WLMO11] Sai Wu, Feng Li, Sharad Mehrotra, and Beng Chin Ooi. Query optimization for massively parallel data processing. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SoCC '11, pages 12:1–12:13, New York, NY, USA, 2011. ACM.

[WOSB11] Xiaodan Wang, Christopher Olston, Anish Das Sarma, and Randal Burns. Coscan: cooperative scan sharing in the cloud. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SoCC '11, pages 11:1–11:12, New York, NY, USA, 2011. ACM.

[Xam] Xamarin. The Mono project. `http://www.mono-project.com`.

[XK09] Y. Xu and P. Kostamaa. Efficient outer join data skew handling in parallel dbms. *Proceedings of the VLDB Endowment*, 2(2):1390–1396, August 2009.

[YAAJ+07] D.G. York, J. Adelman, J.E. Anderson Jr, S.F. Anderson, J. Annis, N.A. Bahcall, JA Bakken, R. Barkhouser, S. Bastian, E. Berman, et al. The sloan digital sky survey: Technical summary. *The Astronomical Journal*, 120(3):1579, 2007.

[YIF+08] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI '08, pages 1–14, Berkeley, CA, USA, December 2008. USENIX Association.

[YLLQ12] Hailong Yang, Zhongzhi Luan, Wenjun Li, and Depei Qian. MapReduce workload modeling with statistical approach. *Journal of Grid Computing*, 10(2):279–310, June 2012. 10.1007/s10723-011-9201-4.

[YYH11] Hsin-Han You, Chun-Chung Yang, and Jiun-Long Huang. A load-aware scheduler for MapReduce framework in heterogeneous cloud environments. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 127–132, New York, NY, USA, 2011. ACM.

[YYTM10] Christopher Yang, Christine Yen, Ceryen Tan, and Samuel Madden. Osprey: Implementing MapReduce-style fault tolerance in a shared-nothing distributed database. In *Proceedings of the IEEE 26th International Conference on Data Engineering*, ICDE '10, pages 657–668, Los Alamitos, CA, USA, 2010. IEEE.

[ZBSS+10] 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 *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM.

[ZBW+12] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Ake Larson, Ronnie Chaiken, and Darren Shakib. SCOPE: parallel databases meet mapreduce. *The VLDB Journal*, 21(5):611–636, October 2012.

[ZGGW11a] Yanfeng Zhang, Qinxin Gao, Lixin Gao, and Cuirong Wang. iMapReduce: A distributed computing framework for iterative computation.

In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, ISPDPSW '11, pages 1112–1121, Washington, DC, USA, May 2011. IEEE Computer Society.

[ZGGW11b] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. PrIter: a distributed framework for prioritized iterative computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SoCC '11, pages 13:1–13:14, New York, NY, USA, 2011. ACM.

[ZKJ+08] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI '08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

List of publications

Major publications

1. Sven Groot, Kazuo Goda, and Masaru Kitsuregawa. A study on workload imbalance issues in data intensive distributed computing. In *Databases in Networked Information Systems*, volume 5999 of *Lecture Notes in Computer Science*, pages 27–32. Springer Berlin Heidelberg, 2010.

2. Sven Groot, Kazuo Goda, and Masaru Kitsuregawa. Towards improved load balancing for data intensive distributed computing. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 139–146, New York, NY, USA, 2011. ACM.

3. Sven Groot. Modeling I/O interference in data intensive Map-Reduce applications. In *Proceedings of the 2012 IEEE/IPSJ 12th International Symposium on Applications and the Internet*, SAINT '12, pages 206–209, Washington, DC, USA, 2012. IEEE Computer Society.

4. Sven Groot, Kazuo Goda, Daisaku Yokoyama, Miyuki Nakano and Masaru Kitsuregawa. Modeling I/O interference for data intensive distributed applications. In *Proceedings of the 2013 ACM Symposium on Applied Computing*, SAC '13, New York, NY, USA, 2013. ACM.

Others

1. Sven Groot, Kazuo Goda, and Masaru Kitsuregawa. Jumbo: a data intensive distributed computation platform – design overview and preliminary experiment –. In *Proceedings of the 72nd National Convention of the Information Processing Society of Japan*, IPSJ '10, Japan, 2010.

2. Sven Groot and Masaru Kitsuregawa. Jumbo: beyond MapReduce for workload balancing. In *Proceedings of the 36th International Conference on Very Large Data Bases (Ph.D. Workshop)*, VLDB '10, Singapore, 2010.

3. Sven Groot, Kazuo Goda, Daisaku Yokoyama, Miyuki Nakano, and Masaru Kitsuregawa. Towards modeling the IO behavior of Map-Reduce applications. In *Proceedings of the 4th Forum on Data Engineering and Information Management*, DEIM '12, Japan, 2012.