Architecting for Performance Clarity in Data Analytics Frameworks

Kay Ousterhout

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2017-158
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-158.html

October 5, 2017

Architecting for Performance Clarity in Data Analytics Frameworks

By

Kay Ousterhout

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Sylvia Ratnasamy, Chair
Professor Scott Shenker
Professor John Chuang

Fall 2017

Architecting for Performance Clarity in Data Analytics Frameworks

Abstract

Architecting for Performance Clarity in Data Analytics Frameworks

by

Kay Ousterhout

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Sylvia Ratnasamy, Chair

There has been much research devoted to improving the performance of data analytics frameworks, but comparatively little effort has been spent systematically identifying the performance bottlenecks of these systems. Without an understanding of what factors are most important to performance, users do not know how to choose a software and hardware configuration to optimize runtime, and developers do not know which optimizations are most important to implement.

This thesis explores how to architect systems for *performance clarity*: the ability to understand where bottlenecks lie and the performance implications of various system changes. First, we focus on incrementally adding performance clarity to current data analytics frameworks. We develop blocked time analysis, a methodology for quantifying performance bottlenecks in parallelized systems, and use it to analyze the Spark framework's performance on two SQL benchmarks and one production workload. Contrary to commonly-held beliefs about performance, we find that (i) CPU (and not I/O) is often the bottleneck, (ii) improving network performance can improve job completion time by at most 2%, and (iii) the causes of most stragglers can be identified.

Blocked time analysis helped to understand performance bottlenecks in today's frameworks, but fell short of enabling users to reason about the impact of potential hardware and software configuration changes. Given the challenges to providing performance clarity in current architectures, the second part of this thesis focuses on a new system architecture built from the ground up for performance clarity. Rather than breaking jobs into tasks that pipeline many resources, as in today's frameworks, we propose breaking jobs into units of work that each use a single resource, called *monotasks*. We demonstrate that explicitly separating the use of different resources simplifies reasoning about performance without sacrificing fast runtimes. Our implementation of monotasks provides job completion times within 9% of Apache Spark, and leads to a model for job completion time that predicts runtime under different hardware and software configurations with at most 28% error for most predictions.

To Mr. Thibodeaux
whose relentless encouragement inspired me
to take my first computer science course

Contents

Acknowledgments

Thanks are due to many people who played a role in this thesis:

To my advisor, Sylvia Ratnasamy, whose ability to provide clarity – whether in writing, determining the next research direction, or interpreting experiment results, and regardless of the research area – guided me through the murkiest moments.

To Scott Shenker, the oracle, whether on issues related to research, teaching, or life.

To Ion Stoica and Matei Zaharia, for introducing me to cloud computing research and for your mentorship through my first conference publication and presentation.

To Mike Freedman, for emailing me after I took his undergraduate course to ask if I was interested in doing my first research project with him. I wasn't sure what research entailed and, at the time, thought networking sounded boring (all of those acronyms!), but a professor had never asked me about working with them before so I decided to give it a try. I might never have started doing research if not for that email.

To the endlesslessly inspirational Jen Rexford, who left me excited about research after every one of our meetings.

To Justine Sherry, who as the first student in the NetSys Lab paved the way for the rest of us, from making sure I had a place to sit on my first day to setting a high standard for writing and presentation quality to helping me typeset this thesis.

To the undergraduate students who I had the honor of working with: Ryan Rasti, Max Wolffe, and Christopher Canel, who were instrumental in completing the experimental work associated with this thesis.

To my officemates in 419 Soda: Aurojit Panda, Shivaram Venkataraman (honorary 419 member), Radhika Mittal, Colin Scott, Amin Tootoonchian, Murphy McCauley, Ethan Jackson, Qifan Pu, Wenting Zheng, Michael Chang, and Aisha Mushtaq. You made graduate school fun, whether we were discussing research challenges or walking to Asha or staying late at Soda Hall before a paper deadline.

To Shivaram Venkataraman and Aurojit Panda, whose friendship and research insights played a particularly significant role in shaping this thesis. Shivaram, you taught me both a more narrow set of skills around using Spark and running EC2 experiments, which were critical to my thesis work, and also the broader lesson that most questions can be answered by reading the code. In addition, your ruthless pursuit of presentation-perfection permanently changed the way I think about presentations. Panda, the breadth of topics on which you are an expert continues to amaze me. Thank you for the help and advice throughout my PhD,

on issues ranging from performance irregularities on EC2 to fixing a flat tire.

To my family and friends. I am deeply grateful for your support and encouragement.

And finally, to my best friend, Patrick Wendell. Thank you for your unwavering belief that I could do it.

Chapter 1

Introduction

Large-scale data analytics frameworks such as Hadoop and Spark have become widely used by businesses to derive value from massive amounts of raw data. These frameworks run on expensive clusters of servers and produce business-critical results, leading to seemingly endless demand for improved efficiency and lower response times. Academia and industry alike have risen to the occasion by producing new frameworks (e.g., Spark [80], Dryad [41], Flink [1]) and myriad optimizations [42, 65, 66, 75, 17, 8, 39, 82, 26, 21, 23, 24, 12, 45, 67, 80, 14, 11, 13, 38, 43, 81] to analyze increasingly large amounts of data in decreasing time.

These efforts have driven down response times at the cost of increased complexity. Performance optimizations typically work by adding functionality to a system, leading to new code paths to reason about and additional parameters to configure the optimization. Furthermore, many performance optimizations work by breaking abstraction barriers. For example, recent optimizations for Spark break the high-level memory abstraction provided by Java, and instead use a library called "Unsafe" to get better performance using C-style memory access [76]. Directly using low level functionality often allows for improved performance, but increases complexity by exposing low-level system details.

As performance optimizations have driven up the complexity of data analytics frameworks, little effort has been spent systematically reasoning about performance. As a result, reasoning about performance is increasingly difficult. For example, two important classes of performance questions are difficult to answer:

What is the bottleneck? Understanding the system bottleneck is critical to future optimization work: without an understanding of the bottleneck, system designers do not know where to focus efforts on improving performance.

How would changes to the system impact performance? Users need to decide between many changes they could make to the system to improve performance; for example, a user might ask which hardware upgrades would have the biggest impact on the performance of her system. Hardware and software configuration changes can significantly improve performance; for example, Venkataraman et al. demonstrated that selecting an appropriate cloud instance type could improve performance by 1.9× without increasing cost [73]. However, existing solutions for determining the ideal configuration rely on sampling all possibilities.

Even making one configuration decision, e.g., what cloud instance type to use, can require sampling dozens of choices; the combinatorial number of settings of different configuration variables makes sampling to determine the best system configuration intractable.

We use the term *performance clarity* to describe the ability to easily answer both of the above questions. Performance clarity has not been well articulated or explored in today's systems, yet is critical to high performance. A lack of performance clarity cripples engineers' ability to devise effective new optimizations, and cripples users' ability to choose between existing optimizations to extract the best performance.

The high-level goal of this thesis is to architect systems to provide performance clarity. Providing performance clarity is difficult because, as we will explain, the use of fine-grained pipelining makes it difficult to reason about task resource use. We begin by focusing on instrumentation, and develop a new technique that uses measurements of blocked times to quantify performance bottlenecks. We use this technique to understand performance of data analytics frameworks, and our findings contradict conventional wisdom about performance bottlenecks. Based on the challenges to providing performance clarity in current systems, the second part of this thesis argues for a new system architecture designed specifically to provide performance clarity.

1.1 The challenge of reasoning about performance

Reasoning about performance is difficult in today's data analytics frameworks. In pursuit of high performance, frameworks employ aggressive parallelism: jobs are divided into many tasks that can execute concurrently on different machines, and each individual task parallelizes CPU, disk, and network using fine-grained pipelining. Fine-grained pipelining is designed to use resources efficiently, but results in ad-hoc resource usage that can change at fine time granularity. This makes it difficult to reason about a job's bottleneck, because even a single task's resource bottleneck can change at fine tine granularity. For example, a task that pipelines disk reads with computation can alternate between bottlenecking on disk, when disk reads contend with other tasks and take a long time to complete, and bottlenecking on CPU, when disk reads are uncontended and finish quickly. Furthermore, tasks may contend at the granularity of the pipelining; e.g., when one task's disk read blocks waiting for a disk write triggered by a different task. This contention is handled by the operating system, making it difficult for the framework to report how different factors contributed to task runtime. As a result, today's frameworks do not provide performance clarity: users cannot easily understand a job's bottleneck or how changes to the system would affect job runtime.

1.2 Instrumenting current frameworks for performance clarity

The first part of this thesis seeks to augment current data analytics frameworks to provide performance clarity. To that end, we propose a new methodology for quantifying performance bottlenecks called blocked time analysis. Blocked time analysis uses extensive white-box logging to measure how long each task spends blocked on a given resource. Taken alone, these per-task measurements allow us to understand straggler causes by correlating slow tasks with long blocked times. Taken together, the per-task measurements for a particular job allow us to simulate how long the job would have taken to complete if the disk or network were infinitely fast, which provides an upper bound on the benefit of optimizing network or disk performance.

We use blocked-time analysis to understand the performance of Spark, a widely used data analytics framework, on two benchmarks and one industry workload. We find that in spite of being highly optimized, current performance is poorly understood. Contrary to the widespread belief that workloads are I/O bound, we find that most workloads are CPU-bound. At the median, network optimizations can improve job completion time by at most 2%, and optimizing or eliminating disk accesses can improve job completion time by at most 19%. We also use blocked time analysis to understand straggler causes. Existing work has had limited success in diagnosing and fixing stragglers, and instead has focused on mitigating the impact of straggler tasks by re-running slow tasks on different machines. Using blocked time analysis, we find that in 75% of queries, we can identify the cause of more than 60% of straggler tasks. These findings call into question the prevailing wisdom about the performance of data analytics frameworks.

Our experience adding blocked time analysis to current data analytics frameworks highlights the difficulty of providing performance clarity in current architectures. First, the differences between our findings and the prevailing wisdom about performance suggest that performance is not currently well understood. Second, augmenting current architectures to reason about performance was difficult: blocked time analysis required a deep understanding of system internals so that we could add extensive white box logging to measure blocked times. This logging is difficult to verify the correctness of, and we found that much of the existing logging in Spark that claimed to instrument blocked times was incomplete. Finally, blocked time analysis meets only one of the requirements of performance clarity – the ability to reason about bottlenecks – but falls short of enabling users to reason about the impact of real system changes like adding additional I/O resources to a cluster.

1.3 Architecting for performance clarity with monotasks

Given the difficulty of adding performance clarity to existing systems, the second part of this thesis explores a more radical re-architecting of data analtyics frameworks to provide performance clarity. We ask: are there alternate system architectures that provide performance

clarity? And do these architectures fundamentally require sacrificing high performance – since many of the optimizations that enabled high performance came hand-in-hand with increases in system complexity?

To provide performance clarity, we propose a departure from today's architectures that rely on multi-resource tasks. Instead, we propose breaking jobs into *monotasks* that each use exactly one of CPU, network, or disk. Each monotask consumes a single resource fully, without blocking on other resources, making it trivial to reason about the resource use of a monotask and, consequently, which resources are in use when. Explicitly separating the use of different resources allows for resource-specific schedulers that can fully utilize the resource (e.g., by running one compute monotask on each CPU core) and queue monotasks to avoid unnecessary contention.

We have used monotasks to implement MonoSpark, an API-compatible replacement for tasks in Apache Spark. MonoSpark does not sacrifice performance, despite eliminating fine-grained pipelining: MonoSpark provides job completion times within 9% of Spark for typical scenarios.

Using monotasks makes it trivial to detect the bottleneck at any point during a job's execution simply by inspecting the length of the queue at the scheduler for each resource, which dramatically simplifies bottleneck detection relative to our blocked time analysis approach.

Looking beyond bottleneck analysis to performance predictions, monotask runtimes can be used to develop a simple model that can predict job completion time under different system configurations. We use the model to predict the runtime of MonoSpark workloads under different hardware configurations (e.g., with a different number or type of disk drives), different software configurations (e.g., with data stored de-serialized and in-memory, rather than on-disk), and with a combination of both hardware and software changes. For "what-if" questions, the monotasks model provides estimates within 28% of the actual job completion time; for example, monotasks correctly predicts the 10× reduction in runtime from migrating a workload to a 4× larger cluster with SSDs instead of HDDs.

Our experience with using monotasks as the basic system building block illustrates that systems *can* provide performance clarity, and can do so without sacrificing high performance.

1.4 Summary of results

To summarize, this thesis makes the following contributions towards providing performance clarity:

- **Blocked time analysis** Blocked time analysis is a new technique for quantifying bottlenecks in highly parallel systems. It uses measurements of how long tasks spent blocked on different resources to determine the best-case improvement to job completion time from optimizing a particular resource.

- **Bottleneck analysis of today's systems** Using blocked time analysis, we quantify performance bottlenecks of Spark for two benchmarks and one production workload.

Contrary to conventional wisdom, we find:

  - **Many workloads are CPU-bound** For the workloads we measured, network optimizations can improve job completion time by a median of at most 2%, and disk optimizations can improve job completion time by a median of at most 19%.
  - **Straggler causes can be identified and fixed** Blocked time analysis can be used to determine the cause of straggler tasks, which allows for more effective mitigation techniques that directly target the underlying cause.

- **Monotasks architecture** Changing the architecture of data analytics frameworks to break today's multi-resource tasks into single-resource monotasks provides performance clarity – including the ability to reason about potential hardware and software configuration changes – without sacrificing performance.

- **Evaluation of Monotasks** We evaluate the performance of a system that uses monotasks rather than today's multi-resource tasks, and find that a system based on monotasks can provide job completion times within 9% of Apache Spark for typical scenarios. We also illustrate that a simple model for job completion time, based on monotask runtimes, can predict the completion time of jobs under different hardware and software configurations with at most 28% error.

1.5 Dissertation plan

This thesis proceeds as follows. In Chapter 2, we describe the architecture of current data analytics frameworks. Chapter 3 describes blocked time analysis, which we use to quantify performance of current data analytics frameworks in Chapter 4. Chapter 5 proposes a new system architecture based on monotasks, and evaluates the performance clarity provided by that architecture. In Chapter 6, we conclude and propose future research directions.

Chapter 2

Background

This chapter describes the architecture of today's data analytics frameworks.

2.1 Framework interface

Data analytics frameworks such as MapReduce [30], Dryad [41], and Spark [80] provide an API to process and generate large datasets. These frameworks have become widely used because they abstract away the details of doing a computation over a large number of machines, and allow the user to express computations using a simple API. Users describe a computation by specifying a function over the dataset; for example, the following Scala code describes a Spark job that reads in a dataset and produces a new dataset with only the lines that begin with "ERROR":

```
lines = spark.textFile("hdfs://path/to/file")
errors = lines.filter(_.startsWith("ERROR"))
```

The framework handles parallelizing the computation over a large cluster of machines, accessing the distributed dataset (in this example, the file located at `hdfs://path/to/file`), applying the specified computation (in this example, selecting only the lines that begin with "ERROR"), and optionally creating a new distributed dataset with the result.

In addition to the general dataflow APIs, frameworks often expose higher-level interfaces that allow users to describe computations in a domain-specific way. For example, many of the workloads used in this thesis rely on SparkSQL, a Spark module that accepts SQL queries, and handles compiling those queries into Spark jobs. Spark also includes a graph module that has an interface for graph computations, as well as a machine learning library.

2.2 Framework architecture

To parallelize computations over many machines, frameworks like MapReduce, Dryad, and Spark use a bulk-synchronous-parallel model where each job is broken into stages, and

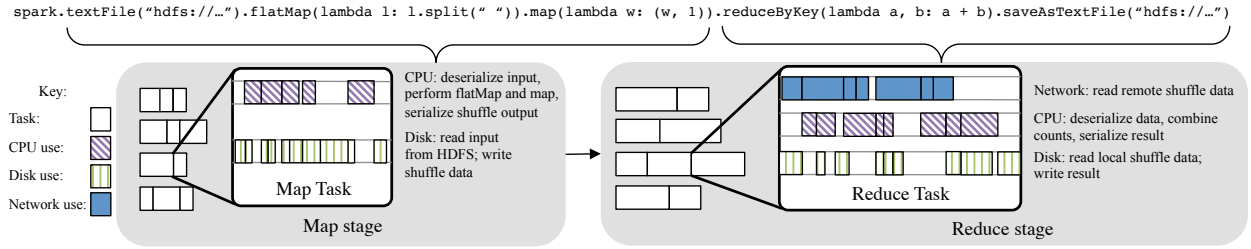


Figure 2.1: Execution of a Spark job, written using Spark's Python API, that performs word count. The job is broken into two stages, and each stage is broken into parallel tasks (shown as white boxes that execute in four parallel slots) that each pipeline use of different resources.

each stage is composed of tasks that execute independently and can be run in parallel. Tasks in a stage all perform the same computation, but do so on different blocks of input data. All synchronization occurs at stage boundaries via a *shuffle*: tasks in one stage write shuffle data to the local disk, and tasks in future stages each read a subset of the shuffle data from all of the previous stage's tasks.

As an example, consider a job that counts the number of occurrences of each word in a large document stored in a distributed file system, as shown in Figure 2.1. This job wold be broken into two stages. The first stage would be composed of map tasks that each read a subset of the input data and output a mapping of words that occur in that subset of the input document to the number of times each word occurs. Each map task would save its output to the disk on the machine where it ran. In the second (reduce) stage, each reduce task would aggregate all of the counts for a subset of the words by reading a subset of the output data from each of the map tasks (corresponding to the words the reduce task is responsible for aggregating counts for), and then summing all of the counts for each word.

MapReduce [30] pioneered this architecture; with MapReduce, each job included exactly one map stage and, optionally, a reduce stage. More complicated queries (e.g., SQL queries) could be executed using a series of MapReduce jobs.

Spark [80] and Dryad [41] were designed to explicitly handle more complicated jobs; in these frameworks, each job is made up of a directed graph of one or more stages. As a result, a single Spark or Dryad job may contain multiple stages of reduce tasks in succession; for example, to compute the result of a SQL query that includes multiple joins. In the remainder of this thesis, we use "map task" to refer to tasks that read blocks of input data stored in a distributed system, and "reduce task" to refer to tasks that read data shuffled from a previous stage of tasks.

Jobs are orchestrated by a central scheduler that decomposes each job into a directed acyclic graph of stages, and assigns the tasks in each stage to worker machines. Schedulers sometimes assign tasks using a fixed number of "slots" on the worker machine, and sometimes usng more complex bin-packing that accounts for the aggregate resource use of each task.

Frameworks use fine-grained pipelining to parallelize use of CPU, network, and disk within each task, as shown in Figure 2.1. In Spark, each task has a single thread that

processes one record at a time: the map task, for example, reads one record from disk, computes on that record, and writes it back to disk. Pipelining is implemented in the background; for example, disk writes typically are written to buffer cache, and the operating system writes to disk asynchronously. Pipelining is not expressed by users as part of the API. For example, in the map stage in Figure 2.1, the user specifies the name of the input file and the transformation to apply to each line of input data, and the framework controls how the file contents are read from disk and passed to the user-specified computation. Frameworks implement pipelining to make tasks complete more quickly by parallelizing resource use.

Chapter 3

Blocked Time Analysis

3.1 Introduction

This chapter focuses on adding performance clarity to existing data analytics frameworks. In particular, we propose a technique that can answer the question "what is the bottleneck?". Determining the bottleneck is difficult for current data analytics frameworks because of the pervasive parallelism described in the previous chapter: jobs are composed of many parallel tasks, and each task uses pipelining to parallelize the use of network, disk, and CPU. One task may be bottlenecked on different resources at different points in execution, and at any given time, tasks for the same job may be bottlenecked on different resources.

We address these challenges with *blocked time analysis*. Blocked time analysis uses extensive white-box logging to measure how long each task spends blocked on a given resource. Taken alone, these per-task measurements allow us to understand straggler causes by correlating slow tasks with long blocked times. Taken together, the per-task measurements for a particular job allow us to simulate how long the job would have taken to complete if the disk or network were infinitely fast, which provides an upper bound on the benefit of optimizing network or disk performance. This methodology does not directly answer the question "what is the bottleneck" because there is typically no single bottleneck; instead, blocked time analysis focuses on quantifying the importance of each resource to a job's completion time.

This remainder of this chapter begins by elaborating on the challenges to understanding performance in current systems (§3.2). Next, we describe how we addressed those challenges with blocked time analysis (§3.3). In Chapter 4, we use blocked time analysis to measure performance of a few benchmark workloads.

The material in this chapter is adapted from previously published work [62].

3.2 Challenges to quantifying bottlenecks

The goal of this chapter is to develop a methodology to understand performance of data analytics workloads; this is a challenging goal for two reasons. First, understanding the

performance of a single task is challenging because tasks pipeline use of multiple resources, as Chapter 2 described. Pipelining results in simultaneous use of different resources, and different resources may be the bottleneck at different times in task execution. This means that naïve approaches to understanding performance are inaccurate. For example, consider a commonly used approach, where the importance of the network is measured by measuring the total amount of time a particular task spent using the network. This approach would typically overestimate the importance of the network: when the network is used in parallel with other resources, speeding up the network at that point would not change the runtime of the task. The fact that tasks use many resources in parallel makes it challenging to measure the impact of a particular resource.

The second challenge to understanding performance is that jobs are composed of many tasks that run in parallel, and each task in a job may have a unique performance profile. For example, one task could encounter network contention and spent most of its execution waiting on the network, while another task could run on a lightly used machine where network requests complete quickly so the task spends most of its time using the CPU. Even if *most* tasks spend little time waiting for the network, if a small number of tasks are delayed waiting for the network and delay the completion time of the job, the network may have a significant impact on the job's end-to-end completion time. This means that any strategy to understand performance needs to consider the job as a whole, rather than looking at the behavior of an average task.

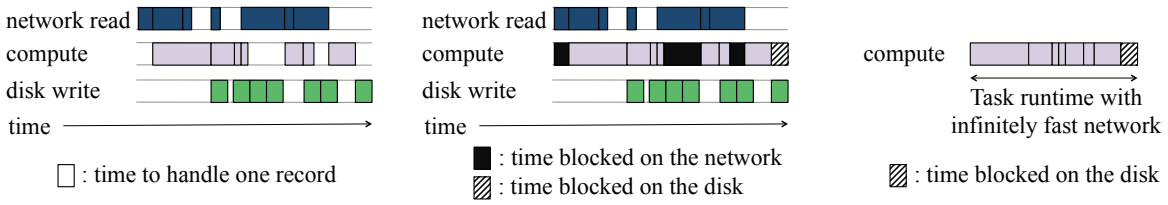
3.3 Blocked time analysis

To make sense of performance, we propose a technique called *blocked time analysis*. The key idea behind blocked time analysis is to focus on the time that tasks spend blocked on a particular resource, because improvements to a particular resource can only speed up the task at points where the task is blocked on that resource. If the task is, for example, doing a disk write in the background, while performing computation, speeding up that disk write will not impact the completion time of the task, because the concurrent computation will still take the same amount of time.

Blocked time analysis quantifies resource bottlenecks by answering the question: how much faster would a job complete if it never blocked on a particular resource? In other words, blocked time analysis describes how much faster a job would get if a particular resource were optimized to be infinitely fast, which represents a best-case scenario of the possible job completion time after implementing a disk or network optimization.

Blocked time analysis lacks the sophistication and generality of general purpose distributed systems performance analysis tools (e.g., [7, 18]), and unlike black-box approaches, requires adding instrumentation within the application. We use blocked-time analysis because unlike existing approaches, it provides a single, easy to understand number to characterize the importance of disk and network use.

Blocked time analysis proceeds in two steps. The first step addresses the first challenge



(a) Pipelined execution of a typical Spark task (b) Blocked times that we measured (c) Task runtime without blocking on network

Figure 3.1: Each Spark task pipelines use of network, CPU, and disk, as shown in (a). To understand the importance of disk and network, we measure times when a task's compute thread blocks on the network or disk, as shown in (b). To determine the best-case task runtime resulting from network (or disk) optimizations, we subtract time blocked on network (or disk), as shown in (c).

to reasoning about performance – that tasks use many resources in parallel – by using white-box logging to measure blocked times. The second step addresses the second challenge to reasoning about performance – that jobs are composed of many parallel tasks – by using simulation.

3.3.1 Measuring per-task blocked times

The first step in blocked time analysis quantifies the performance of a particular task by focusing on blocked time: time the task spends blocked on the network or the disk (shown in Figure 3.1(b)). We focus on blocked time from the perspective of the compute thread, because it provides a single vantage point from which to measure. In Spark, the task's computation runs as a single thread, whereas network requests are issued by multiple threads that operate in the background, and disk I/O is pipelined by the OS, outside of the Spark process. We focus on blocked time, rather than measuring all time when the task is using the network or the disk, because network or disk performance improvements cannot speed up parts of the task that execute in parallel with network or disk use. Blocked time analysis can only be used to understand the potential improvement from optimizing network or disk; §3.3.3 explains why we cannot use blocked time analysis to quantify the potential improvement from optimizing use of the CPU.

After measuring the amount of time each task spends blocked on a particular resource, we subtract this blocked time from the task's original runtime to determine how quickly the task could have completed if that resource were infinitely fast. Figure 3.1(c) shows an example of using this process to determine how quickly a task could have completed if the network were infinitely fast.

This methodology may overestimate the importance of a resource for a particular task, because blocked times may overlap with use of other resources. For example, in Figure 3.1(c), during some of the times when the CPU was blocked on the network, the task was also using disk to write output data. If the network had been faster, some of the time spent blocked

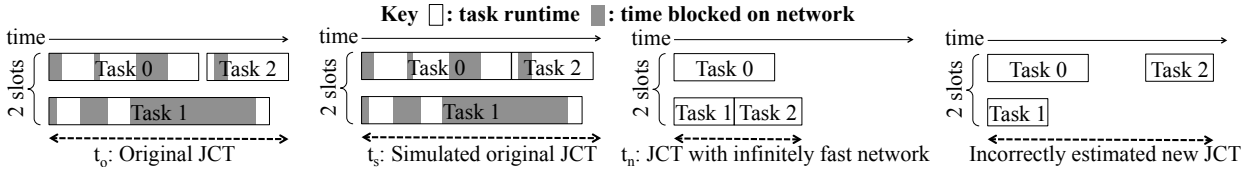


Figure 3.2: To compute a job's completion time (JCT) without time blocked on network, we subtract the time blocked on the network from each task, and then simulate how the tasks would have been scheduled, given the number of slots used by the original runtime and the new task runtimes. We perform the same computation for disk.

on the network would instead have been spent blocked on the disk. As a result, blocked time analysis may overestimate the improvement from optimizing a particular resource. We emphasize throughout this chapter that blocked time analysis represents the best-case improvement from optimizing a particular resource.

3.3.2 Simulating job completion time

The second step in blocked time analysis uses simulation to understand how shorter completion times for each task would affect the completion time of the job as a whole. The simulation replays the execution of the scheduling of the job's tasks, based on the number of slots used by the original job and the new task runtimes. Figure 3.2 shows a simple example. The example on the far right illustrates why we need to replay execution rather than simply use the original task completion times minus blocked time: that approach underestimates improvements because it does not account for multiple waves of tasks (task 2 should start when the previous task finishes, not at its original start time) and does not account for the fact that tasks might have been scheduled on different machines given different runtimes of earlier tasks (task 2 should start on the slot freed by task 1 completing). We replay the job based only on the number of slots used by the original job, and do not take into account locality constraints that might have affected the scheduling of the original job. This simplifying assumption does not significantly impact the accuracy of our simulation: at the median, the time predicted by our simulation is within 4% of the runtime of the original job. The ninety-fifth percentile error is at most 7% for the benchmark workloads and 27% for the production workload.[1] In order to minimize the effect of this error on our results, we always compare the simulated time without network (or disk) to the *simulated* original time. For example, in the example shown in Figure 3.2, we would report the improvement as t_n/t_s (i.e., time with infinitely fast network divided by simulated original time), rather than as t_n/t_o (time with infinitely fast network divided by the original job completion time). This focuses our results on the impact of a particular resource rather than on error introduced by our simulation.

[1] The production workload has higher error because we don't model pauses between stages that occur when SparkSQL is updating the query plan. If we modeled these pauses, the impact of disk and network I/O would be lower.

3.3.3 Why can't blocked time analysis be used to understand CPU use?

While blocked time analysis can be used to understand the importance of network and disk, unfortunately it does not help to quantify the importance of computation. This is because we cannot easily measure when the task is only using the CPU: when a task is not blocked on network or disk, it is difficult to measure whether the network or disk I/O are being performed on behalf of the task in the background. I/O is often performed by the operating system and out of view of the data analytics framework; for example, disk writes are often placed in the buffer cache, and the operating system writes the data to disk in the background. In Spark, which was the focus of our instrumentation, all of the tasks on a particular machine run in a single process, which further complicates understanding when a particular task is using network or disk resources: even if the disk on a machine is currently in use, it is difficult to determine which task that disk use corresponds to. As a result of these challenges, we do not use blocked time analysis to understand the importance of the CPU. In the next chapter, Section 4.5.3 instead relies on coarser-grained utilization measurements to understand CPU use.

3.4 Conclusion

This chapter developed a new technique, blocked time analysis, for quantifying performance bottlenecks in data analytics frameworks. The key challenge in developing blocked time analysis is the use of fine-grained pipelining: understanding task resource use is difficult because a task's resource profile may change at the granularity of the pipelining. Blocked time analysis addressed this challenge by focusing on the time tasks spent blocked on network or disk use, and using those measurements to determine the best-case improvement to job completion time as a result of optimizing a resource. As we will demonstrate in the next chapter, the upper bounds reported by blocked time analysis are a useful metric for understanding the importance of different resources to job. However, blocked time analysis cannot be used to determine *how much* a particular resource needs to be optimized to access the best-case improvement: would a 20% speedup be sufficient, or would the resource need to get five or ten times as fast to get the potential improvement reported by blocked time analysis? In Chapter 5, we explore a new system architecture designed specifically for performance clarity, and that can answer what-if questions about potential speedups to a resource.

Chapter 4

Making Sense of Performance in Today's Frameworks

4.1 Introduction

This chapter uses the technique proposed in the previous chapter, blocked time analysis, to understand performance in today's analytics frameworks.

Today's analytics frameworks are commonly believed to have three performance properties, which have each motivated significant optimization effort:

1. **The network is a bottleneck.** This has motivated work on a range of network optimizations, including load balancing across multiple paths, leveraging application semantics to prioritize traffic, aggregating data to reduce traffic, isolation, and more [42, 65, 66, 75, 17, 8, 39, 82, 26, 21, 23, 22, 24].

2. **The disk is a bottleneck.** This has led to work on using the disk more efficiently [67] and caching data in memory [12, 45, 74, 80].

3. **Straggler tasks significantly prolong job completion times and have largely unknown underlying causes.** This has driven work on mitigation using task speculation [81, 14, 11, 13] or running shorter tasks to improve load balancing [61]. Researchers have been able to identify and target a small number of underlying straggler causes such as data skew [14, 43, 38] and popularity skew [10].

Most of this work focuses on a particular aspect of the system in isolation, leaving us without a comprehensive understanding of which factors are most important to the end-to-end performance of data analytics workloads.

This chapter uses blocked time analysis to understand Spark's performance on two industry benchmarks and one production workloads. In studying the applicability of the three aforementioned claims to these workloads, we find:

1. **Network optimizations can only reduce job completion time by a median of at most 2%.** The network is not a bottleneck because much less data is sent over the network than is transferred to and from disk. As a result, network I/O is mostly irrelevant to overall performance, even on 1Gbps networks.

2. **Optimizing or eliminating disk accesses can only reduce job completion time by a median of at most 19%.** CPU utilization is typically much higher than disk utilization; as a result, engineers should be careful about trading off I/O time for CPU time by, for example, using more sophisticated serialization and compression techniques.

3. **Optimizing stragglers can only reduce job completion time by a median of at most 10%, and in 75% of queries, we can identify the cause of more than 60% of stragglers.** Blocked-time analysis illustrates that the two leading causes of Spark stragglers are Java's garbage collection and time to transfer data to and from disk. We found that targeting the underlying cause of stragglers could reduce non-straggler runtimes as well, and describe one example where understanding stragglers in early experiments allowed us to identify a bad configuration that, once fixed, reduced job completion time by a factor of two.

These results question the prevailing wisdom about the performance of data analytics frameworks. By necessity, our study does not look at a vast range of workloads nor a wide range of cluster sizes, because the ability to add finer-grained instrumentation to Spark was critical to our analysis. As a result, we cannot claim that our results are broadly representative. However, the fact that the prevailing wisdom about performance is so incorrect on the workloads we do consider suggests that there is much more work to be done before our community can claim to understand the performance of data analytics frameworks.

To facilitate performing blocked time analysis on a broader set of workloads, we have added almost all[1] of our instrumentation to Spark and made our analysis tools publicly available. We have also published the detailed benchmark traces that we collected so that other researchers may reproduce our analysis or perform their own [57].

The remainder of this chapter begins by describing the instrumentation we added to Spark to enable blocked time analysis (§4.2), the workloads we studied (§4.3), and our experimental setup (§4.4). Next, we explore the importance of disk I/O (§4.5), the importance of network I/O (§4.6), and the importance and causes of stragglers (§4.7); in each of these sections, we discuss the relevant related work and contrast it with our results. We explore the impact of cluster and data size on our results in §4.8. We end with reflections on what has changed in the two years since this work was completed (§4.9).

This chapter expands on previously published work [62].

[1]Some of our logging needed to be added outside of Spark, as was described in Chapter 3, because it could not be implemented in Spark with sufficiently low overhead.

4.2 Instrumentation

Obtaining the measurements described in §3.3.1 requires extensive white box logging. We added this logging to Spark so that we could understand performance of Spark workloads; this required significant improvements to the instrumentation in Spark, as well as adding instrumentation to Hadoop Distributed Filesystem (HDFS), which Spark workloads often use to read and write distributed datasets. While some of the instrumentation required was already available in Spark, our detailed performance analysis revealed that existing logging was often incorrect or incomplete [53, 54, 55, 69, 56, 58]. Where necessary, we fixed existing logging and pushed the changes upstream to Spark.

We found that cross validation was crucial to validating that our measurements were correct. In addition to instrumentation for blocked time, we also added instrumentation about the CPU, network, and disk utilization on the machine while the task was running (per-task utilization cannot be measured in Spark, because all tasks run in a single process). Utilization measurements allowed us to cross validate blocked times; for example, by ensuring that when tasks spent little time blocked on I/O, CPU utilization was correspondingly high. In many cases, we were able to identify shortcomings in existing logging for time blocked on disk because tasks reported little time blocked on disk, but had very high disk utilization and low CPU utilization while running, which implied that the time blocked on disk was incorrectly reported.

As part of adding instrumentation, we measured Spark's performance before and after the instrumentation was added to ensure the instrumentation did not add to job completion time. To ensure logging did not affect performance, we sometimes had to add logging outside of Spark. For example, to measure time spent reading input data, we needed to add logging in the HDFS client when the client reads large "packets" of data from disk, to ensure that timing calls were amortized over a relatively time-consuming read from disk.[2] Adding the logging in Spark, where records are read one at a time from the HDFS client interface, would have degraded performance.

4.3 Workloads

This section describes the workloads we used for blocked time analysis, and production traces that we used to sanity check our results.

4.3.1 Benchmark workloads

Our analysis centers around fine-grained instrumentation of two benchmarks and one production workload running on Spark, summarized in Table 4.1.

[2]This logging is available in a modified version of Hadoop at https://github.com/kayousterhout/hadoop-common/tree/2.0.2-instrumented

Workload name	Total queries	Cluster size	Data size
Big Data Benchmark [72], Scale Factor 5 (BDBench)	50 (10 unique queries, each run 5 times)	40 cores (5 machines)	60GB
TPC-DS [70], Scale Factor 100	140 (7 users, 20 unique queries)	160 cores (20 machines)	17GB
TPC-DS [70], Scale Factor 5000	260 (13 users, 20 unique queries)	160 cores (20 machines)	850GB
Production	30 (30 unique queries)	72 cores (9 machines)	tens of GB

Table 4.1: Summary of workloads run. In addition to these workloads, we study one larger workload in §4.8, and three network intensive workloads in §4.6.4.

The big data benchmark (BDBench) [72] was developed to evaluate the differences between analytics frameworks and was derived from a benchmark developed by Pavlo et al. [64]. The input dataset consists of HTML documents from the Common Crawl document corpus [4] combined with SQL summary tables generated using Intel's Hadoop benchmark tool [79]. The benchmark consists of four queries including two exploratory SQL queries, one join query, and one page-rank-like query. The first three queries have three variants that each use the same input data size but have different result sizes to reflect a spectrum between business-intelligence-like queries (with result sizes that could fit in memory on a business intelligence tool) and ETL-like queries with large result sets that require many machines to store. We run the queries in series and run five iterations of each query. We use the same configuration that was used in published results [72]: we use a scale factor of five (which was designed to be run on a cluster with five worker machines), and we run two versions of the benchmark. The first version operates on data stored in-memory using SparkSQL's columnar cache (cached data is not replicated) and the second version operates on data stored on-disk using Hadoop Distributed File System (HDFS), which triply replicates data for fault-tolerance.

Our second benchmark is a variant of the Transaction Processing Performance Council's decision-support benchmark (TPC-DS) [70]. The TPC-DS benchmark was designed to model multiple users running a variety of decision-support queries including reporting, interactive OLAP, and data mining queries. All of the users run in parallel; each user runs the queries in series in a random order. The benchmark models data from a retail product supplier about product purchases. We use a subset of 20 queries that was selected in an existing industry benchmark that compares four analytics frameworks [32]. Similar to with the big data benchmark, we run two variants. The first variant stores data on-disk using Parquet [2], a compressed columnar storage format that is the recommended storage format for high performance with Spark SQL, and uses a scale factor of 5000. The second, in-memory variant uses a smaller scale factor of 100; this small scale factor is necessary because SparkSQL's cache is not well optimized for the type of data used in the TPC-DS benchmark, so while the

data only takes up 17GB in the compressed on-disk format, it occupies 200GB in memory. We run both variants of the benchmark on a cluster of 20 machines.

The final Spark workload described by the results in this chapter is a production workload from Databricks that uses their cloud product [5] to submit ad-hoc Spark queries. Input data for the queries includes a large fact table with over 50 columns. The workload includes a small number of ETL queries that read input data from an external file system into the memory of the cluster; subsequent queries operate on in-memory data and are business-intelligence-style queries that aggregate and summarize data. Data shown in future graphs breaks the workload into the in-memory and on-disk components. For confidentiality reasons, further details of the workload cannot be disclosed.

4.3.2 Production traces

Where possible, we sanity-checked our results with coarse-grained analysis of traces from Facebook, Google, and Microsoft. The Facebook trace includes 54K jobs run during a contiguous period in 2010 on a cluster of thousands of machines (we use a 1-day sample from this trace). The Google data includes all MapReduce jobs run at Google during three different one month periods in 2004, 2006, and 2007, and the Microsoft data includes data from an analytics cluster with thousands of servers on a total of eighteen different days in 2009 and 2010. While our analysis would ideally have used *only* data from production analytics workloads, all data made available to us includes insufficient instrumentation to compute blocked time. For example, the default logs written by Hadoop (available for the Facebook cluster) include only the total time for each map task, but do not break map task time into how much time was spent reading input data and how much time was spent writing output data. This has forced researchers to rely on estimation techniques that can be inaccurate, as we show in §4.6.5. Therefore, our analysis begins with a detailed instrumentation workloads we ran ourselves on Spark, but in most cases, we demonstrate that our high-level takeaways are compatible with production data.

4.4 Cluster setup

For the benchmark workloads, we ran experiments using a cluster of Amazon EC2 m2.4xlarge instances, which each have 68.4GB of memory, two disks, and eight vCPUs. Our experiments use Apache Spark version 1.2.1 and Hadoop version 2.0.2. Spark runs queries in long-running processes, meaning that production users of Spark will run queries in a JVM that has been running for a long period of time. To emulate that environment, before running each benchmark, we ran a single full trial of all of the benchmark queries to warm up the JVM. For the big data benchmark, where only one query runs at a time, we cleared the OS buffer cache on all machines before launching each query, to ensure that input data is read from disk. While our analysis focuses on one cluster size and data size for each benchmark, we illustrate that scaling to larger clusters does not significantly impact

our results in §4.8.

The production workload ran on a 9-machine cluster with 250GB of memory; further details about the hardware configuration are proprietary. The cluster size and hardware is representative of Databricks' users.

4.5 How important is disk I/O?

Previous work has suggested that reading input data from disk can be a bottleneck in analytics frameworks; for example, the Spark research paper describes speedups of 40× for generating a data analytics report as a result of storing input and output data in memory using Spark, compared to storing data on-disk and using Hadoop for computation [80]. The PACMan paper reported reducing average job completion times by 53% as a result of caching data in-memory [12]. The assumption that many data analytics workloads are I/O bound has driven numerous research proposals (e.g., Themis [67], Tachyon [45]) and the implementation of in-memory caching in industry [74]. Based on this work, our expectation was that time blocked on disk would represent the majority of job completion time.

4.5.1 How much time is spent blocked on disk I/O?

Using blocked time analysis, we compute the potential improvement in job completion time if tasks did not spend any time blocked on disk I/O.[3] This involved measuring time blocked on disk I/O at four different points in task execution:

1. Reading input data stored on-disk (this only applies for the on-disk workloads; in-memory workloads read input from memory).

2. Writing shuffle data to disk. Spark writes all shuffle data to disk, even when input data is read from memory.

3. Reading shuffle data from a remote disk. This time includes both disk time (to read data from disk) and network time (to send the data over the network). Network and disk use is tightly coupled and thus challenging to measure separately; we measure the total time as an *upper bound* on the improvement from optimizing disk performance.

4. Writing output data to local disk and two remote disks (this only applies for the on-disk workloads). Again, the time to write data to remote disks includes network time as well; we measure both the network and disk time, making our results an upper bound on the improvement from optimizing disk.

[3] This measurement includes only time blocked on disk requests, and does *not* include CPU time spent deserializing byte buffers into Java objects. This time is sometimes considered disk I/O because it is a necessary side effect of storing data outside of the JVM; we consider only disk hardware performance here, and discuss serialization time separately in §4.5.5.

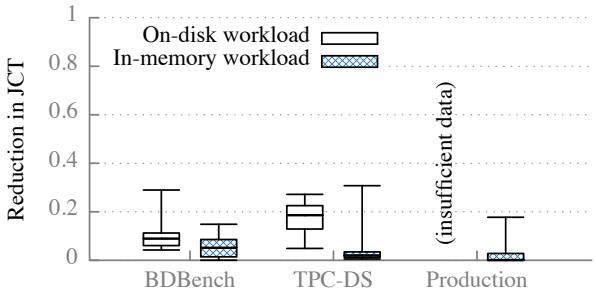


Figure 4.1: Improvement in job completion time (JCT) as a result of eliminating all time spent blocked on disk I/O. Boxes depict 25th, 50th, and 75th percentiles; whiskers depict 5th and 95th percentiles.

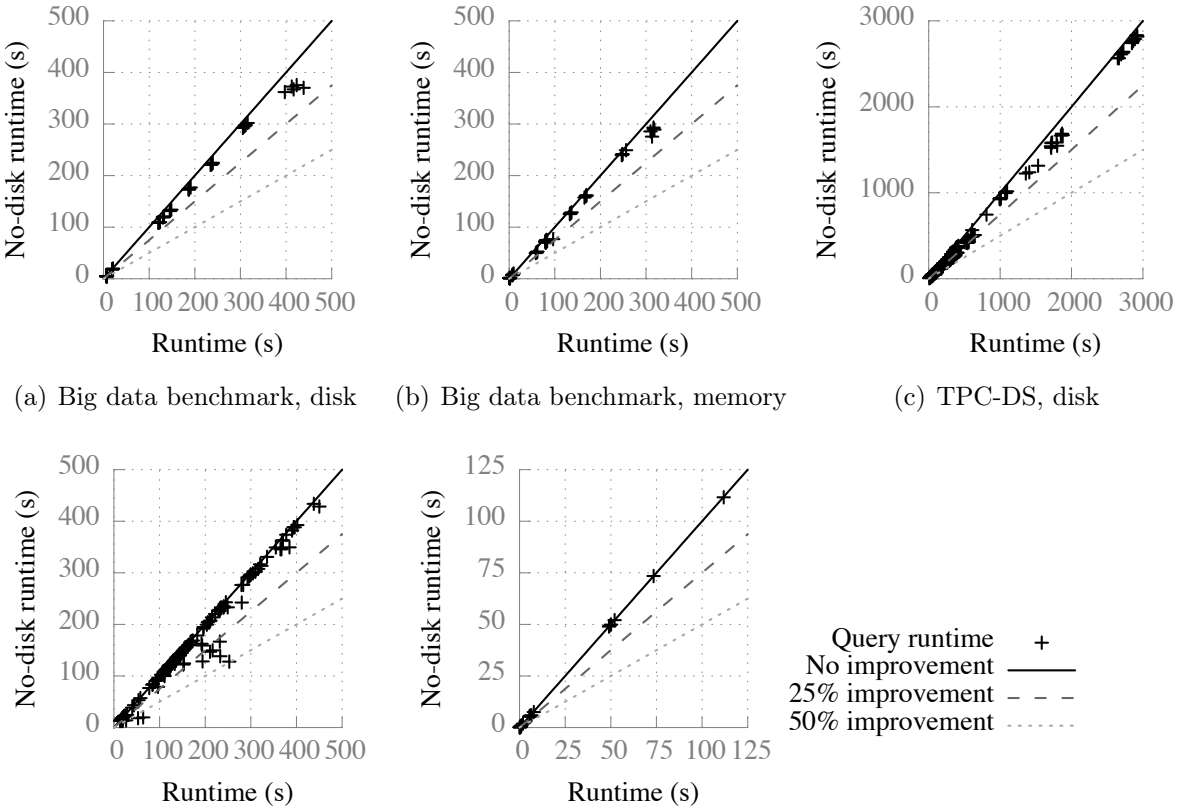
Using blocked time analysis, we find that the median improvement from eliminating all time blocked on disk is at most 19% across all workloads, as shown in Figure 4.1. The y-axis in Figure 4.1 describes the relative reduction in job completion time; a reduction of 0.1 means that the job could complete 10% faster as a result of eliminating time blocked on the disk. The figure illustrates the distribution over jobs, including all trials of each job in each workload. Boxes depict 25th, 50th, and 75th percentiles; whiskers depict 5th and 95th percentiles. The variance stems from the fact that different jobs are affected differently by disk. The on-disk queries in the production workload are not shown in Figure 4.1 because, as described in §4.2, instrumenting time to read input data required adding instrumentation to HDFS, and this was not possible to do for the production cluster. We show the same results in Figure 4.2, which instead plots the absolute runtime originally and the absolute runtime once time blocked on disk has been eliminated. The scatter plots illustrate that long jobs are not disproportionately affected by time reading data from disk.

For in-memory workloads, the median improvement from eliminating all time blocked on disk is 2-5%; the fact that this improvement is non-zero is because even in-memory workloads store shuffle data on disk.

A median improvement in job completion time of 19% is a respectable improvement in runtime, but is lower than we expected for workloads that read input data and store output data on-disk. The following two subsections make more sense of this number, first by considering the effect of our hardware setup, and then by examining alternate metrics to put this number in the context of how tasks spend their time.

4.5.2 How does hardware configuration affect these results?

Hardware configuration impacts blocked time: tasks that run on machines with fewer disks relative to the number of CPU cores would have spent more time blocked on disk, and vice versa. In its hardware recommendations for users purchasing Hadoop clusters, one vendor recommends machines with at least a 1:3 ratio of disks to CPU cores [51]. In 2010, Facebook's Hadoop cluster included machines with between a 3:4 and 3:1 ratio of disks to CPU cores [19]. Our machines, with a 1:4 ratio of disks to CPU cores, have relatively under-



(a) Big data benchmark, disk (b) Big data benchmark, memory (c) TPC-DS, disk

(d) TPC-DS, in-memory (e) Production, in-memory

Figure 4.2: Comparison of original runtimes of Spark jobs to runtimes when all time blocked on the disk has been eliminated.

provisioned I/O capacity based on these recommendations. As a result, I/O may appear *more* important in our measurements than it would in the wild; in other words, our results on time blocked on disk still represent an upper bound on the importance of I/O.

The second aspect of our setup that affects results is the number of concurrent tasks run per machine; we run one task per core, consistent with the Spark default.

4.5.3 How does disk utilization compare to CPU utilization?

Our result that eliminating time blocked on disk I/O can only improve job completion time by a median of at most 19% suggests that jobs may be CPU bound. Unfortunately, as described in 3.3.3, we cannot use blocked time analysis to understand the importance of compute time, so we instead examine CPU and disk utilization. Figure 4.3 plots the distribution across all tasks in the big data benchmark and TPC-DS benchmark of the CPU utilization compared to disk utilization. For these workloads, the plot illustrates that on average, queries are more CPU bound than I/O bound. Hence, tasks are likely blocked

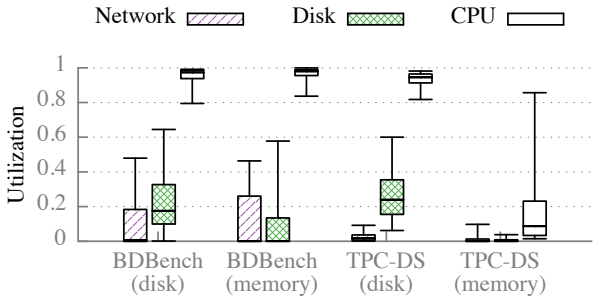


Figure 4.3: Average network, disk, and CPU utilization while tasks were running. CPU utilization is the total fraction of non-idle CPU milliseconds while the task was running, divided by the eight total vCPUs on the machine. Network utilization is the bandwidth usage divided by the machine's 1Gbps available bandwidth. All utilizations are obtained by reading the counters in the /proc file system at the beginning and end of each task. The distribution is across all tasks in each workload, weighted by task duration.

waiting on computation to complete more often than they are blocked waiting for disk I/O. On clusters with more typical ratios of disk to CPU use, the disk utilization will be even lower relative to CPU utilization.

4.5.4 Sanity-checking our results against production traces

The fact that the disk is not the bottleneck in our workloads left us wondering whether our workloads were unusually CPU bound, which would mean that our results were not broadly representative. Unfortunately, production data analytics traces available to us do not include blocked time or disk utilization information. However, we were able to use aggregate statistics about the CPU and disk use of jobs to compare the I/O demands of our workloads to the I/O demands of larger production workloads. In particular, we measured the I/O demands of our workloads by measuring the ratio of data transferred to disk to non-idle CPU seconds:

$$\text{MB / CPU second} = \frac{\text{Total MB transferred to/from disk}}{\text{Total non-idle CPU seconds}}$$

This metric is imperfect because it looks at the CPU and disk requirements of the job as a whole, and does not account for variation in resource usage during a job. Nonetheless, it allows us to understand how the aggregate disk demands of our workloads compare to large-scale production workloads. In particular, if production workloads had a higher rate of data transferred per non-idle CPU seconds, it would suggest that our workloads were unusually CPU-bound, and that production workloads might see a larger improvement from disk optimizations.

Using this metric, we found that the I/O demands of the three workloads we ran do not differ significantly from the I/O demands of production workloads. Figure 4.4 illustrates that for the benchmarks and production workload we instrumented, the median MB / CPU

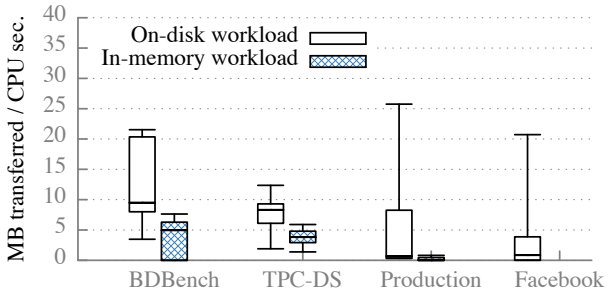


Figure 4.4: Average megabytes transferred to or from disk per non-idle CPU second for all jobs we ran. The median job transfers less than 10 megabytes to/from disk per CPU second; given that effective disk throughput was approximately 90 megabytes per second per disk while running our benchmarks, this demand can easily be met by the two disks on each machine.

	Aug. '04	Mar. '06	Sep. '07
Map input data (TB) [31]	3,288	52,254	403,152
Map output data (TB) [31]	758	6,743	34,774
Reduce output data (TB) [31]	193	2,970	14,018
Task years used [31]	217	2,002	11,081
Total data transferred to/from disk (TB)	5383	74,650	514,754
Avg. MB transferred to/from disk per task second (MB/s)	.787	1.18	1.47
Avg. Mb sent over the network per task second (Mbps)	1.34	1.61	1.44

Table 4.2: Disk use for all MapReduce jobs run at Google.

second is less than 10. Figure 4.4 also illustrates results for the trace from Facebook's Hadoop cluster. The MB / CPU second metric is useful in comparing to Hadoop performance because it relies on the volume of data transferred to disk and the CPU milliseconds to characterize the job's demand on I/O, so abstracts away many inefficiencies in Hadoop's implementation (for example, it abstracts away the fact that a CPU-bound query may take much longer than the CPU milliseconds used due to inefficient resource use). The number of megabytes transferred to disk per CPU second is lower for the Facebook workload than for our workloads: the median is just 3MB/s for the Facebook workload, compared to a median of 9MB/s for the big data benchmark on-disk workload and 8MB/s for the TPC-DS workload.

We also examined aggregate statistics published about Microsoft's Cosmos cluster and Google's MapReduce cluster, shown in Tables 4.2 and 4.3. Unlike the Facebook trace, the Google and Microsoft statistics do not include a measurement of the CPU time spent by jobs, and instead quote the total time that tasks were running, aggregated across all jobs [29]. In computing the average rate at which tasks transfer data to and from disk, we assume that the input data is read once from disk, intermediate data is written once (by map tasks that generate the data) and read once (by reduce tasks that consume the data), and output data is written to disk three times (assuming the industry standard triply-replicated output data).

	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	Jan
Input Data (PB) [14]	12.6	22.7	14.3	18.7	22.8	25.3	25.0	18.6	21.5
Intermediate Data (PB) [14]	0.66	1.22	0.67	0.76	0.73	0.86	0.68	0.72	1.99
Compute (years) [14]	49.1	88.0	51.6	60.6	73.0	84.1	88.4	96.2	79.5
Avg. MB read/written per task second	8.99	9.06	9.61	10.58	10.54	10.19	9.46	6.61	10.16
Avg. Mb shuffled per task second	3.41	3.52	3.29	3.18	2.54	2.59	1.95	1.90	6.35

Table 4.3: Disk use for a cluster with tens of thousands of machines, running Cosmos. Compute (years) describes the sum of runtimes across all tasks [15]. The data includes jobs from two days of each month; see [14] for details.

As shown in Tables 4.2 and 4.3, based on this estimate, Google jobs transfer an average of 0.787 to 1.47 MB/s to disk,[4] and Microsoft jobs transfer an average of 6.61 to 10.58 MB/s to disk. These aggregate numbers reflect an estimate of *average* I/O use so do not reflect tail behavior, do not include additional I/O that may have occurred (e.g., to spill intermediate data), and are not directly comparable to our results because unlike the CPU milliseconds, the total task time includes time when the task was blocked on network or disk I/O. The takeaway from these results should not be the precise value of these aggregate metrics, but rather that sanity checking our results against production traces does not lead us to believe that production workloads have dramatically different I/O requirements than the workloads we measure.

4.5.5 Why isn't disk I/O more important?

We were surprised at the results in §4.5.3 given the oft-quoted mantra that I/O is typically the bottleneck, and also the fact that fundamentally, the computation done in data analytics job is often very simple. For example, queries 1a, 1b, and 1c in the big data benchmark select a filtered subset of a table. Given the simplicity of that computation, we would not have expected the query to be CPU bound. One reason for this result is that today's frameworks often store compressed data (in increasingly sophisticated formats, e.g. Parquet [2]), which trades CPU time for I/O time. We found that if we instead ran queries on uncompressed data, most queries became I/O bound. A second reason that CPU time is large is an artifact of the decision to write Spark in Scala, which is based on Java: after being read from disk, data must be deserialized from a byte buffer to a Java object. Figure 4.5 illustrates the distribution of the total non-idle CPU time used by queries in the big data benchmark

[4] For the Google traces, the aggregate numbers are skewed by the fact that, at the time, a few MapReduce jobs that consumed significant resources were also very CPU intensive (in particular, the final phase of the indexing pipeline involved significant computation). These jobs also performed some additional disk I/O from within the user-defined map and reduce functions [29]. We lack sufficient information to quantify these factors, so they are not included in our estimate of MB/s.

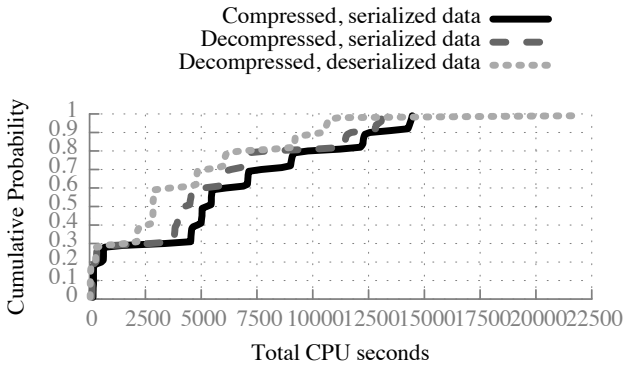


Figure 4.5: Comparison of total non-idle CPU milliseconds consumed by the big data benchmark workload, with and without compression and serialization.

under 3 different scenarios: when input data, shuffle data, and output data are compressed and serialized; when input data, shuffle data, and output data are not deserialized but are decompressed; and when input and output data are stored as deserialized, in-memory objects (shuffle data must still be serialized in order to be sent over the network). The distribution includes one data point for each trial of each query, and each point on the CDF describes what percentage of queries (shown on the Y-axis) had total CPU seconds of at most the value on the X-axis. The CDF illustrates that for some queries, as much as half of the CPU time is spent deserializing and decompressing data. This result is consistent with Figure 9 from the Spark paper [80], which illustrated that caching deserialized data significantly reduced job completion time relative to caching data that was still serialized.

Spark's relatively high CPU time may also stem from the fact that Spark was written Scala, as opposed to a lower-level language such at C++. For one query that we re-wrote in C++, we found that the CPU time reduced by a factor of more than 2×. Existing work has illustrated that writing analytics in C++ instead can significantly improve performance [27], and the fact that Google's MapReduce is written in C++ is an oft-quoted reason for its superior performance.

4.5.6 Are these results inconsistent with past work?

Prior work has shown significant improvements as a result of storing input data for analytics workloads in memory. For example, Spark was demonstrated to be 20× to 40× faster than Hadoop [80]. A close reading of that paper illustrates that much of that improvement came not from eliminating disk I/O, but from other improvements over Hadoop, including eliminating serialization time.

The PACMan work described improvements in average job completion time of more than a factor of two as a result of caching input data [12], which, similar to Spark, seems to potentially contradict our results. The 2× improvements were shown for two workloads. The first workload was based on the Facebook trace, but because the Facebook trace does not include enough information to replay the exact computation used by the jobs, the au-

thors used a mix of compute-free (jobs that do not perform any computation), sort, and word count jobs [15]. This synthetic workload was generated based on the assumption that analytics workloads are I/O bound, which was the prevailing mentality at the time. Our measurements suggest that jobs are not typically I/O bound, so this workload may not have been representative. The second workload was based on a Bing workload (rewritten to use Hive), where the 2× improvement represented the difference in reading data from a serialized, on-disk format compared to reading de-serialized, in-memory data. As discussed in §4.5.5, serialization times can be significant, so the 2× improvement likely came as much from eliminating serialization as it did from eliminating disk I/O.

4.5.7 Summary

We found that job runtime cannot improve by more than 19% as a result of optimizing disk I/O. To shed more light on this measurement,we compared disk and CPU utilization while tasks were running, and found that CPU utilization is typically close to 100% whereas median disk utilization is at most 25%. One reason for the relatively high use of CPU by the analytics workloads we studied is deserialization and compression; the shift towards more sophisticated serialization and compression formats has decreased the I/O and increased the CPU requirements of analytics frameworks. Because of high CPU times, optimizing hardware performance by using more disks, using flash storage, or storing serialized in-memory data will yield only modest improvements to job completion time; caching deserialized data has the potential for much larger improvements due to eliminating deserialization time.

Serialization and compression formats will inevitably evolve in the future, rendering the numbers presented in this chapter obsolete. The takeaway from our measurements should be that CPU usage is *currently* much higher than disk use, and that detailed performance instrumentation like our blocked time analysis is critical to navigating the tradeoff between CPU and I/O time going forward.

4.6 How important is the network?

Researchers have used the argument that data-intensive application performance is closely tied to datacenter network performance to justify a wide variety of network optimizations [42, 65, 66, 75, 17, 8, 39, 82, 26, 21, 23, 22, 24]. We therefore expected to find that optimizing the network could yield significant improvements to job completion time.

4.6.1 How much time is spent blocked on network I/O?

To understand the importance of the network, we first use blocked time analysis to understand the largest possible improvement from optimizing the network. As shown in Figure 4.6, none of the workloads we studied could improve by a median of more than 2% as a result of optimizing network performance. We did not use especially high bandwidth machines in getting this result: the m2.4xlarge instances we used have a 1Gbps network link.

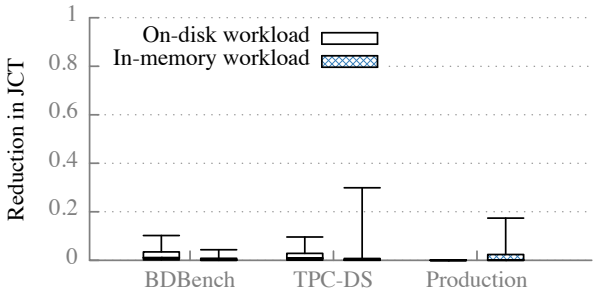


Figure 4.6: Improvement in job completion time as a result of eliminating all time blocked on network.

Our blocked time instrumentation for the network included time to read shuffle data over the network, and for on-disk workloads, the time to write output data to one local machine and two remote machines. Both of these times include disk use as well as network use, because disk and network are interlaced in a manner that makes them difficult to measure separately. As a result, as with all of our blocked time measurements, 2% represents an upper bound on the possible improvement from network optimizations.

To shed more light on the network demands of the workloads we ran, Figure 4.3 plots the network utilization along with CPU and disk utilization. Consistent with the fact that blocked times are very low, median network utilization is lower than median disk utilization and much lower than median CPU utilization for all of the workloads we studied.

4.6.2 Sanity-checking our results against production traces

We were surprised at the low impact of the network on job completion time, given the large body of work targeted at improving network performance for analytics workloads. Similar to what we did in §4.5.3 to understand disk performance, we computed the network data sent per non-idle CPU second, to facilitate comparison to large-scale production workloads, as shown in Figure 4.7. Similar to what we found for disk I/O, the Facebook workload transfers less data over the network per CPU second than the workloads we ran. Thus, we expect that running our blocked time analysis on the Facebook workload would yield smaller potential improvements from network optimizations than for the workloads we ran. Tables 4.2 and 4.3 illustrate this metric for the Google and Microsoft traces, using the machine seconds or task seconds in place of CPU milliseconds as with the disk results.[5] For Google, the average megabits sent over the network per machine second ranges from 1.34 to 1.61; Microsoft network use is higher (1.90-6.35 megabits are shuffled per task second) but still far below the network use seen in our workload.Thus, this sanity check does not lead us to believe that production workloads have dramatically different network requirements than the workloads we measured.

[5]The Microsoft data does not include output size, so network data only includes shuffle data.

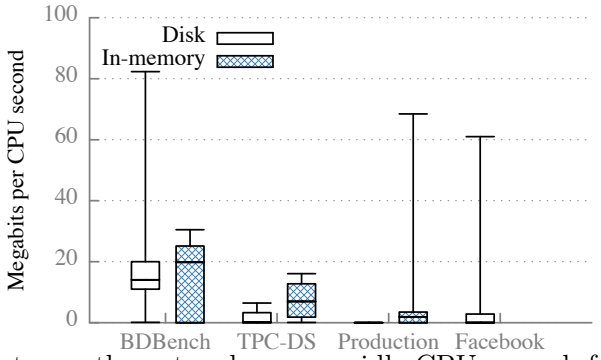


Figure 4.7: Megabits sent over the network per non-idle CPU second, for the workloads we instrumented and for a production workload from Facebook.

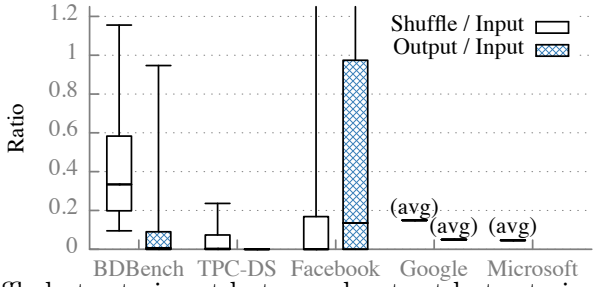


Figure 4.8: Ratio of shuffle bytes to input bytes and output bytes to input bytes. The median ratio of shuffled data to input data is less than 0.35 for all workloads, and the median ratio of output data to input data is less than 0.2 for all workloads.

4.6.3 Why isn't the network more important?

One reason that network performance is relatively unimportant is that the amount of data sent over the network is often much less than the data transferred to disk, because analytics queries often shuffle and output much less data than they read. Figure 4.8 plots the ratio of shuffle data to input data and the ratio of output data to input data across our benchmarks (the production workload did not have sufficient instrumentation to capture these metrics), for the Facebook trace, and for the Microsoft and Google aggregate data (averaged over all of the samples). Across all workloads, the amount of data shuffled is less than the amount of input data, by as much as a factor of 5-10, which is intuitive considering that data analysis often centers around aggregating data to derive a business conclusion. In evaluating the efficacy of optimizing shuffle performance, many papers have used workloads with ratios of shuffled data to input data of 1 or more, which these results demonstrate is not widely representative.

4.6.4 When is the network important?

While we found a small potential improvement to job completion time as a result of optimizing the network, some prior work (e.g., GraphX [35]) has shown significant improve-

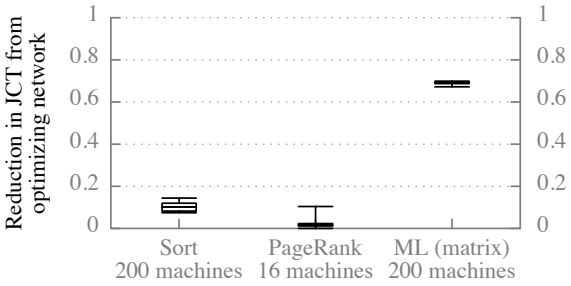


Figure 4.9: Improvement in job completion time as a result of eliminating all time blocked on the network, for three workloads known to be network-intensive

ments to job completion time as a result of optimizing network use. This section explores workloads where the network is known to be important, to characterize the circumstances under which the network has a significant impact on job completion time. Figure 4.9 shows three such workloads. The first workload sorts key-value pairs using 200 machines. The sort workload does not perform any filtering or aggregation, so the amount of shuffled data and output data is equal to the amount of input data, which we expected to lead to significant opportunities to speed up job completion time as a result of optimizing the network. The median improvement to job completion time as a result of optimizing the network was 10.2%, which is higher than for the workloads we studied, but not as high as we had expected. We profiled the job while it was running to understand why tasks were not spending more time blocked on the network, and found that tasks spent a significant amount of time serializing and deserializing data, consistent with what we concluded when studying the importance of the disk in §4.5.5.

The second workload shown in Figure 4.9 is the PageRank workload used in the GraphX paper [35]. This workload uses GraphX, a Spark module that compiles graph computations into Spark jobs. Graph workloads are commonly known to be network intensive because they typically need to shuffle significant amounts of data, but similar to the sort workload, with blocked time analysis we found that the median best-case improvement from improvements to the network was 1.5%. Again, profiling revealed that the job spent significant amounts of time serializing and deserializing data.[6] The GraphX paper reported nearly 2× improvements to iteration runtimes as a result of sending less data over the network (Figure 6 in [35]). These improvements stemmed from eliminating both the network time to send the data *and* the CPU time to serialize and deserialize the additional data. Our measurements suggest that this improvement came primarily from reducing the CPU time rather than from reducing the network time.

The final workload we measured is a machine learning workload that uses a series of

[6]Studies of frameworks with more optimized serialization have found the network to have more significant impact when running PageRank workloads [48, 49]; these results are discussed in more detail in the conclusion to this chapter (§4.9).

matrix multiplications to perform a least squares fit; for this workload, the reduction to job completion time as a result of optimizing the network is as high as 69%. This workload has three properties that, in combination, mean that the network could benefit significantly from improvements to the network. First, the computation done by the workload relies on optimized, native code, so completes quickly. Second, the data is stored as arrays of doubles, which means that it is very inexpensive to serialize and deserialize. Finally, in order to perform the matrix multiplications, the workload needs to shuffle significantly more data than the size of the input data. This workload illustrates that if a workload shuffles large amounts of data *and* has very efficient computation and serialization, the network will be important. If future versions of Spark or future frameworks use more optimized computation and serialization, similar to this workload, the network will become important for a broader class of workloads.

4.6.5 Are these results inconsistent with past work?

Some past work has reported high effects of the network on job completion time, based on studying traces of production workloads. Existing traces typically do not include sufficient metrics to precisely measure the impact of the network, and we demonstrate that estimation techniques used to understand the network overestimated its importance. For example, Sinbad [21] asserted that writing output data can take a significant amount of time for analytics jobs, but used a heuristic to understand time to write output data: it defined the write phase of a task as the time from when the shuffle completes until the completion time of the task. This metric is an estimate, which was necessary because Hadoop does not log time blocked on output writes separately from time spent in computation. Unfortunately, this estimate can overestimate the time spent writing output data, because it does not account for any computation (including serializing output data) that occurs after the shuffle completes. We instrumented Hadoop to log time spent writing output data and ran the big data benchmark (using Hive to convert SQL queries to Map Reduce jobs) and compared the result from our detailed instrumentation to the estimation previously used. Unfortunately, as shown in Figure 4.10, the previously used metric significantly overestimates time spent writing output data for the big data benchmark. This suggests that the importance of the network was significantly overestimated.

A second problem with past estimations of the importance of the network is that they have conflated inefficiencies in Hadoop with network performance problems. One commonly cited work quotes the percent of job time spent in shuffle, measured using a Facebook Hadoop workload [23]. We repeated this measurement using the Facebook trace, shown in Table 4.4. Previous measurements looked at the fraction of jobs that spent a certain percent of time in shuffle (i.e., the first two lines of Table 4.4); by this metric, 16% of jobs spent more than 75% of time shuffling data. We dug into this data and found that a typical job that spends more than 75% of time in its shuffle phase takes tens of seconds to shuffle kilobytes of data, suggesting that framework overhead and not network performance is the bottleneck. We also found that only 1% of all jobs spend less than 4 seconds shuffling data, which suggests that

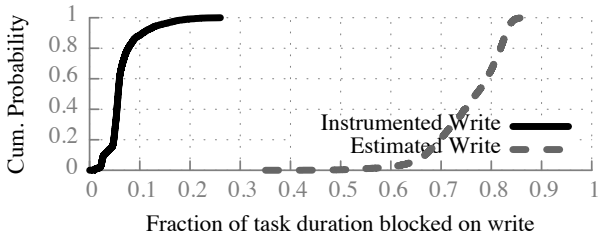


Figure 4.10: Cumulative distribution across tasks (weighted by task duration) of the fraction of task time spent writing output data, measured using our fine grained instrumentation and estimated using the technique employed by prior work [21]. The previously used metric far overestimates time spent writing output data.

Shuffle Dur.	< 25%	25-49%	50-74%	>= 75%
% of Jobs	46%	20%	18%	16%
% of time	91%	7%	2%	1%
% of bytes	83%	16%	1%	0.3%

Table 4.4: The percent of jobs, task time, and bytes in jobs that spent different fractions of time in shuffle for the Facebook workload.

the Hadoop shuffle includes fixed overheads to, for example, communicate with the master to determine where shuffle data is located. For such tasks, shuffle is likely bottlenecked on inefficiencies and fixed overheads in Hadoop, rather than by network bottlenecks.

Table 4.4 includes data not reported in prior work: for each bucket, we compute not only the percent of jobs that fall into that bucket, but also the percent of total time and percent of total bytes represented by jobs in that category. While 16% of jobs spend more than 75% of the time in shuffle, these jobs represent only 1% of the total bytes sent across the network, and 0.3% of the total time taken by all jobs. This further suggests that the jobs reported as spending a large fraction of time in shuffle are small jobs for which the shuffle time is dominated by framework overhead rather than by network performance.

4.6.6 Summary

We found that, for the three workloads we studied, network optimizations can only improve job completion time by a median of at most 2%. One reason network performance has little effect on job completion time is that the data transferred over the network is a subset of data transferred to disk, so jobs bottleneck on the disk before bottlenecking on the network. We found this to be true in a cluster with 1Gbps network links; in clusters with 10Gbps or 100Gbps networks, network performance will be even less important.

Past work has found much larger improvements from optimizing network performance for two reasons. First, some past work has focused only on workloads where shuffle data is equal to the amount of input data, which we demonstrated is not representative of typical workloads. Second, some of this work conflated network time with serialization time, as we found in §4.6.4. Finally, some past work relied on estimation to understand trace data,

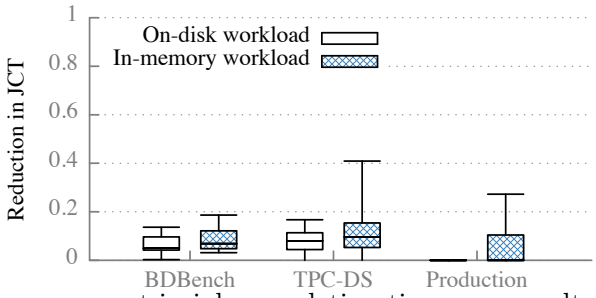


Figure 4.11: Potential improvement in job completion time as a result of eliminating all stragglers. Results are shown for the on-disk and in-memory versions of each benchmark workload.

which led to misleading conclusions about the importance of the network.

4.7 The role of stragglers

A straggler is a task that takes much longer to complete than other tasks in the stage. Because the stage cannot complete until all of its tasks have completed, straggler tasks can significantly delay the completion time of the stage and, subsequently, the completion time of the higher-level query. Past work has demonstrated that stragglers can be a significant bottleneck; for example, stragglers were reported to delay average job completion time by 47% in Hadoop clusters and by 29% in Dryad clusters [11]. Existing work has characterized some stragglers as being caused by data skew, where a task reads more input data than other tasks in a job, but otherwise does not characterize what causes some tasks to take longer than others [14, 43, 38]. This inability to explain the cause of stragglers has typically led to mitigation strategies that replicate tasks, rather than strategies that attempt to understand and eliminate the root cause of long task runtimes [11, 13, 81, 61].

4.7.1 How much do stragglers affect job completion time?

To understand the impact of stragglers, we adopt the approach from [11] and focus on a task's inverse progress rate: the time taken for a task divided by amount of input data read. Consistent with that work, we define the potential improvement from eliminating stragglers as the reduction in job completion time as a result of replacing the progress rate of every task that is slower than the median progress rate with the median progress rate. We use the methodology described in §3.3.2 to compute the new job completion time given the new task completion times. Figure 4.11 illustrates the improvement in job completion time as a result of eliminating stragglers in this fashion; the median improvement from eliminating stragglers is 5-10% for the big data benchmark and TPC-DS workloads, and lower for the production workloads, which had fewer stragglers.

4.7.2 Are these results inconsistent with prior work?

Some prior work has reported the effect of stragglers to be similar to what we found in our study. For example, based on a Facebook trace, Mantri described a median improvement of 15% as a result of eliminating all stragglers. Mantri had a larger overall improvements when deployed in production that stemmed not only from eliminating stragglers, but also from eliminating costly recomputations [14].

Other work has reported larger potential improvements from eliminating stragglers: Dolly, for example, uses the same metric we used to understand the impact of stragglers, and found that eliminating stragglers could reduce job completion time by 47% for a Facebook trace and 29% for a Bing trace [11]. This difference may be due to the larger cluster size or greater heterogeneity in the studied traces. However, the difference can also be partially attributed to framework differences. For example, Spark has much lower task launch overhead than Hadoop, so Spark often breaks jobs into many more tasks, which can reduce the impact of stragglers [61]. Stragglers would have had a much larger impact for the workloads we ran if all of the tasks in each job had run as a single wave; in this case, the median improvement from eliminating stragglers would have been 23-41%. We also found that stragglers have become less important as Spark has matured. When running the same benchmark queries with an older version of Spark, many stragglers were caused by poor disk performance when writing shuffle data. Once this problem was fixed, the importance of stragglers decreased. These improvements to Spark may make stragglers less of an issue than previously observed, even on large-scale clusters.

4.7.3 Why do stragglers occur?

Prior work investigating straggler causes has largely relied on traces with coarse grained instrumentation; as a result, it has been able to attribute stragglers to data skew and high resource utilization [14, 78], but otherwise has been unable to explain why stragglers occur. Our instrumentation allows us to describe the cause of more than 60% of stragglers in 75% of the queries we ran.

In examining the cause of stragglers, we follow previous work and define a straggler as a task with inverse progress rate greater than 1.5× the median inverse progress rate for the stage. Unlike the previous subsection, we do not look at all tasks that take longer than the median progress rate, in order to focus on situations where there was a significant anomaly in a task's execution.

Many stragglers can be explained by the fact that the straggler task spends an unusually long amount of time in a particular part of task execution. We characterize a straggler as caused by X if it would not have been considered a straggler had X taken zero time for all of the tasks in the stage. We use this methodology to attribute stragglers to scheduler delay (time taken by the scheduler to ship the task to a worker and to process the task completion message), HDFS disk read time, shuffle write time, shuffle read time, and Java's garbage collection (which can be measured using Java's `GarbageCollectorMXBean` interface).

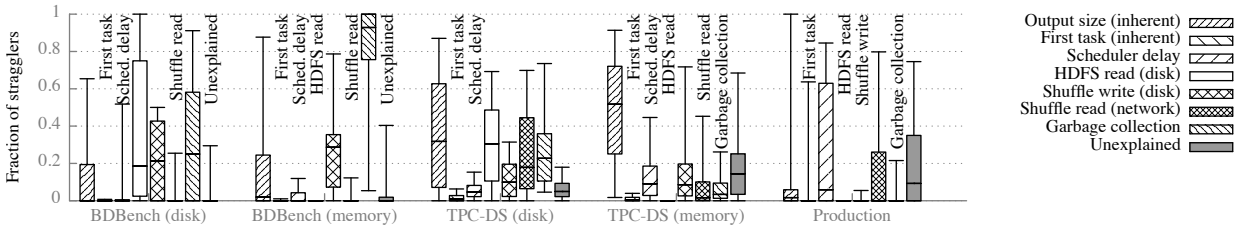


Figure 4.12: Our improved instrumentation allowed us to explain the causes behind most of the stragglers in the workloads we instrumented. This plot shows the distribution across all queries of the fraction of the query's stragglers that can be attributed to a particular cause.

We attribute stragglers to two additional causes that require different methodologies. First, we attribute stragglers to output skew by computing the progress rate based on the amount of output data processed by the task instead of the amount of input data, and consider a straggler caused by output skew if the task is a straggler based on input progress rate but not based on output progress rate. Second, we find that some stragglers can be explained by the fact that they were among the first tasks in a stage to run on a particular machine. This effect may be caused by Java's just-in-time compilation (jit): Java runtimes (e.g., the HotSpot JVM [52]) optimize code that has been executed more than a threshold number of times. We consider stragglers to be caused by the fact that they were a "first task" if they began executing before any other tasks in the stage completed on the same machine, and if they are no longer considered stragglers if compared to other "first tasks."

For each of these causes, Figure 4.12 plots the fraction of stragglers in each query explained by that cause. The distribution arises from differences in straggler causes across queries; for example, for the on-disk big data benchmark, in some jobs, all stragglers are explained by the time to read data from HDFS, whereas in other jobs, most stragglers can be attributed to garbage collection.

The graph does not point to any one dominant of stragglers, but rather illustrates that straggler causes vary across workloads and even within queries for a particular workload. However, common patterns are that garbage collection can cause most of the stragglers for some queries, and many stragglers can be attributed to long times spent reading to or writing from disk (this is not inconsistent with our earlier results showing a 19% median improvement from eliminating disk: the fact that some straggler tasks are caused by long times spent blocked on disk does not necessarily imply that overall, eliminating time blocked on disk would yield a large improvement in job completion time). Another takeaway is that many stragglers are caused by inherent factors like output size and running before code has been jitted, so cannot be alleviated by straggler mitigation techniques.

4.7.4 Improving performance by understanding stragglers

Understanding the root cause behind stragglers provides ways to improve performance by mitigating the underlying cause. In our early experiments, investigating straggler causes led

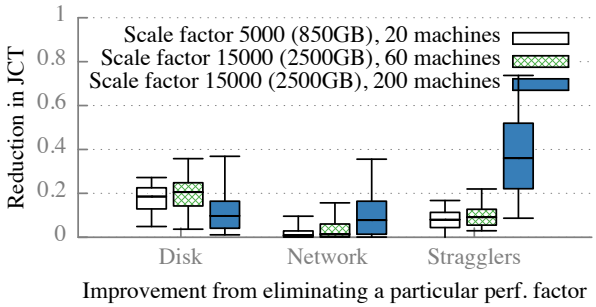


Figure 4.13: Improvement in job completion time as a result of eliminating disk I/O, eliminating network I/O, and eliminating stragglers, each shown for two different trials of the on-disk TPC-DS workload using different scale factors and cluster sizes.

us to find that the default file system on the EC2 instances we used, ext3, performs poorly for workloads with large numbers of parallel reads and writes, leading to stragglers. By changing the filesystem to ext4, we fixed stragglers *and* reduced median task time, reducing query runtime for queries in the big data benchmark by $17 - 58\%$. Many of the other stragglers we observed could potentially be reduced by targeting the underlying cause, for example by allocating fewer objects (to target GC stragglers) or consolidating map output data into fewer files (to target shuffle write stragglers). Understanding these causes allows for new straggler mitigation techniques beyond duplicating slow tasks.

4.8 How does scale affect results?

The results in this chapter have focused on one cluster size for each of the benchmarks run. To understand how our results might change in a much larger cluster, we ran the TPC-DS on-disk workload on two larger clusters: one with three times as many machines and using three times more input data, and one using ten times as many machine and three times more input data. Figure 4.13 compares our key results on the larger clusters to the results from the 20-machine cluster described in the remainder of this chapter. For the 60-machine cluster, the potential improvements from eliminating disk I/O, eliminating network I/O, and eliminating stragglers are comparable to the corresponding improvements on the smaller cluster. The 200-machine cluster has two key differences. First, the 200-machine cluster has a larger potential improvement from optimizing the network. We studied this improvement and found that the issue is not network bandwidth: network utilization is comparable to with the 20 and 60 machine clusters. Instead, with the larger number of machines, at any given time typically at least one machine is performing garbage collection. This causes the shuffle to take longer, because garbage collection on any one machine can cause the shuffle to pause. The second difference is that stragglers are more important for the 200-machine cluster. We found that for half of the jobs, at least 45% of the stragglers were due to scheduler delay, because the scheduler sometimes could not keep up with the larger

cluster size. These stragglers will not benefit from existing mitigation techniques that re-run the tasks elsewhere; in fact, re-running tasks will lead to more stragglers from scheduler delay, because the scheduler will have to handle a larger number of tasks. These differences provide insight into different problems that arise in larger clusters, but do not contradict the findings in this chapter that optimizing the network will have minimal impact on job completion time and that most straggler causes can be identified.

4.9 Conclusion and reflections

This chapter undertook a detailed performance study of three workloads, and found that for those workloads, jobs are often bottlenecked on CPU and not I/O, network performance has little impact on job completion time, and many straggler causes can be identified and fixed.

When we published this work [62], we noted that our findings should not be taken as the last word on performance of analytics frameworks, because they represent only one snapshot in time, and a small set of workloads. Shortly after we concluded our study, Spark developers began a project to optimize Spark's CPU use [76], driven in part by our work demonstrating that the CPU is often a bottleneck. Spark has gotten significantly more CPU-efficient as a result of this optimization work, and recent work has shown sizable improvements from optimizing the network in newer versions of Spark [71]. Other researchers measured the importance of the network in Timely Dataflow in Rust, a system with more optimized computation than our version of Spark, and similarly found the network to be important to job completion time [48, 49].

In highly optimized systems, the bottleneck will be in a constant state of change, because developers continuously work to improve performance of the current bottleneck. Given that the only constant is change, the takeaway from our work should be the importance of systems that provide performance clarity. The more recent work mentioned above studied the importance of the network by actually upgrading the network and then measuring the change in job completion time. Our systems should provide easier ways for users to understand which factors are most important to performance, so that the resource bottleneck is not a topic of debate, and so that researchers and practitioners alike can understand how best to focus performance improvements.

Chapter 5

Monotasks: Architecting for Performance Clarity

5.1 Introduction

Chapters 3 and 4 focused on a technique, blocked time analysis, that adds performance clarity to existing system architectures. While blocked time analysis was useful for answering simple questions about performance bottlenecks, it does not allow users to answer common practical questions about how to optimize for the best performance. For example:

What hardware should I run on? Is it worth it to get enough memory to cache on-disk data? How much will upgrading the network from 1Gbps to 10Gbps improve performance?

What software configuration should I use? Should I store compressed or uncompressed data? How much work should be assigned concurrently to each machine?

Why did my workload run so slowly? Is hardware degradation leading to poor performance? Is performance affected by contention from other users?

Effectively answering the above questions can lead to significant performance improvements; for example, Venkataraman et al. demonstrated that selecting an appropriate cloud instance type could improve performance by 1.9× without increasing cost [73]. Yet answering these questions in existing systems remains difficult. Moreover, expending significant effort to answer these questions *once* is not sufficient: users must continuously re-evaluate as software optimizations, hardware improvements, and workload changes shift the bottleneck.

This chapter argues that architecting for performance clarity – making it easy to understand where bottlenecks lie and the performance implications of various system changes – should be an integral part of system design. Systems that simplify reasoning about performance enable users to determine what configuration parameters to set and what hardware to use to optimize runtime.

To provide performance clarity, we propose building systems in which the basic unit of scheduling consumes only one resource. In the remainder of this chapter, we apply this principle to large-scale data analytics frameworks. We present an architecture that decomposes data analytics jobs into *monotasks* that each use exactly one of CPU, disk, or network. Each

resource has a dedicated scheduler that schedules the monotasks for that resource. This design contrasts with today's frameworks, which distribute work over machines by dividing it into tasks that each parallelize the use of CPU, disk, and network.

Decoupling the use of different resources into monotasks simplifies reasoning about performance. With current frameworks, the use of fine-grained pipelining to parallelize CPU, disk, and network results in ad-hoc resource use that can change at fine time granularity. This makes it difficult to reason about a job's bottleneck, because even a single task's resource bottleneck can change on short time horizons, as described in §3.2. Furthermore, tasks may contend at the granularity of the pipelining; e.g., when one task's disk read blocks waiting for a disk write triggered by a different task. This contention is handled by the operating system, making it difficult for the framework to report how different factors contributed to task runtime. In contrast, each monotask consumes a single resource fully, without blocking on other resources, making it trivial to reason about the resource use of a monotask and, as a result, the resource use of the job as a whole. Controlling each resource with a dedicated scheduler allows that scheduler to fully utilize the resource and queue monotasks to control contention.

We used monotasks to implement MonoSpark, an API-compatible version of Apache Spark that replaces current multi-resource tasks with monotasks. MonoSpark does not sacrifice performance, despite eliminating fine-grained resource pipelining: on three benchmark workloads, MonoSpark provides job completion times within 9% of Spark for typical scenarios (§5.5). In some cases, MonoSpark outperforms Spark, because per-resource schedulers allow MonoSpark to avoid resource contention and under utilization that occur as a result of the non-uniform resource use during the lifespan of Spark tasks.

Using monotasks to schedule resources leads to a simple model for job completion time based on the runtimes of each type of monotask. We use the model to predict the runtime of MonoSpark workloads under different hardware configurations (e.g., with a different number or type of disk drives), different software configurations (with data stored de-serialized and in-memory, rather than on-disk), and with a combination of both hardware and software changes (§5.6). For most "what-if" questions, the monotasks model provides estimates within 28% of the actual job completion time; for example, monotasks correctly predicts the 10× reduction in runtime from migrating a workload to a 4× larger cluster with SSDs instead of HDDs.

Using monotasks also makes bottleneck analysis trivial. Monotask runtimes can easily be used to determine the bottleneck resource, and we demonstrate that the monotasks model can be used for the same analysis we used blocked time analysis to perform in Chapter 4.

Finally, using monotasks leads to new opportunities for performance optimizations. For example, MonoSpark can automatically determine the ideal task concurrency, which users are required to specify in existing frameworks. We illustrate one example where using monotasks to automatically determine the ideal concurrency improves job completion time by 30% relative to configuring a job's resource use (§5.7).

This chapter expands on previously published work [60, 59].

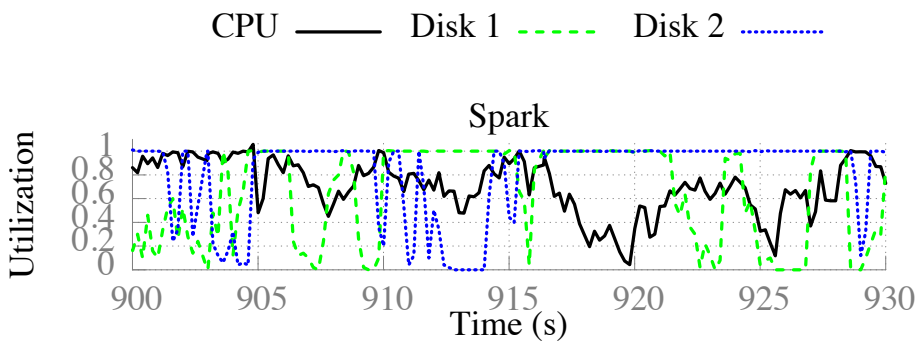


Figure 5.1: As a result of fine-grained pipelining, resource utilization during Spark jobs is non-uniform. This example illustrates the utilization during a 30 second period when 8 tasks were running concurrently, and when the bottleneck changes between CPU and disk.

5.2 The challenge of reasoning about performance

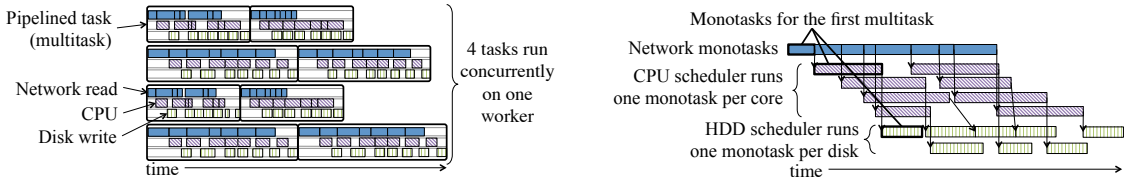
As described in Chapter 2, data analytics frameworks like MapReduce [30], Dryad [41], and Spark [80] use fine-grained pipelining to parallelize the use of CPU, network, and disk within each task. This makes reasoning about performance difficult for three reasons:

Tasks have non-uniform resource use. A task's resource profile may change at fine time granularity as different parts of the pipeline become a bottleneck. For example, the reduce task in Figure 2.1 bottlenecks on CPU, network, and disk at different points during its execution. Figure 5.1 illustrates this effect during a thirty-second period of time when 8 concurrent Spark tasks were running on a machine. During this period, the resource utilization oscillates between being bottlenecked on CPU and being bottlenecked on one of the disks, as a result of fine-grained changes in each task's resource usage.

Concurrent tasks on a machine may contend. Each use of network, CPU, or disk may contend with other tasks running on the same machine. For example, when multiple tasks simultaneously issue disk reads or writes for the same disk, those requests will contend and take longer to complete. This occurs in Figure 5.1 at 920 seconds, when all eight concurrent tasks block waiting on requests from the two disks.

Resource use occurs outside the control of the analytics framework. Resource use is often triggered by the operating system. For example, data written to disk is typically written to the buffer cache. The operating system, and not the framework, will eventually flush the cache, and this write may contend with later disk reads or writes.

Together, these three challenges make reasoning about performance difficult. Consider a user who asks a question like "how much more quickly would my job have run if it used twice as many disks". To answer this question, she would need to walk through a task's execution at the level of detail of each fine-grained resource use. For each fine-grained use of network, CPU, and disk, the user would need to determine whether the time for that resource use would change in the new scenario, factoring in how timing would be affected by the resource use of other tasks on the same machine – which would each need to be modeled at similarly



(a) Execution as today's multi-resource tasks.

(b) Execution as monotasks. Arrows represent dependencies.

Figure 5.2: Execution of eight multitasks on a single worker machine. With current frameworks, shown in (a), each multitask parallelizes reading data over the network, filtering some of the data (using the CPU), and writing the result to disk. Using monotasks, shown in (b), each of today's multitasks is decomposed into a DAG of monotasks, each of which uses exactly one resource. Per-resource schedulers regulate access to each resource.

fine time granularity. The complexity of this process explains the lack of simple models for job completion time.

Existing approaches to reasoning about performance have addressed this challenge in two different ways. One approach was the blocked time analysis we described in Chapter 3: we added instrumentation to measure when tasks block on different resources, and used that instrumentation to answer simple questions about resource bottlenecks. Unfortunately, blocked time analysis cannot be used for more sophisticated what-if questions. A second approach treats the analytics framework as a black box, and uses machine learning techniques to build a new performance model for each workload by running the workload (or a representative subset of the workload) repeatedly under different configurations (e.g., Ernest [73] and CherryPick [9]). These models can answer what-if questions, but require offline training for each new workload. The complexity of these approaches is mandated by the challenges of reasoning about performance in current frameworks.

Given the importance of the ability to reason about performance and the challenges to doing so today, we explore architectural approaches to providing performance clarity. We ask: would a different system architecture make it easy to reason about performance? And would such an architecture require a simplistic approach that sacrifices fast job completion times?

5.3 Monotasks architecture

We explore a system architecture that provides performance clarity by using single-resource units of work called monotasks. Our focus is to explore how systems can provide performance clarity, and as a result we do not focus on optimizing our implementation to maximize performance. Using monotasks enables many new scheduling optimizations, and we leave exploration of such optimizations to future work.

5.3.1 Design

The monotasks design replaces the fine-grained pipelining used in today's tasks – henceforth referred to as *multi*tasks – with statistical multiplexing across *mono*tasks that each use a single resource. The design is based on four principles:

Each monotask uses one resource. Jobs are decomposed into units of work called monotasks that each use exactly one of CPU, network, and disk. As a result, the resource use of each task is uniform and predictable.

Monotasks execute in isolation. To ensure that each monotask can fully utilize the underlying resource, monotasks do not interact with or block on other monotasks during their execution.

Per-resource schedulers control contention. Each worker machine has a set of schedulers that are each responsible for scheduling monotasks on one resource. These resource schedulers are designed to run the minimum number of monotasks necessary to keep the underlying resource fully utilized, and queue remaining monotasks. For example, the CPU scheduler runs one monotask per CPU core. This design makes resource contention "visible" as the queue length for each resource.

Per-resource schedulers have complete control over each resource. To ensure that the per-resource schedulers can control contention for each resource, monotasks avoid optimizations that involve the operating system triggering resource use. For example, disk monotasks flush all writes to disk, to avoid situations where the OS buffer cache contends with other disk monotasks.

Figure 5.2 compares execution of multitasks on a single worker machine with current frameworks, shown in (a), with how those multitasks would be decomposed into monotasks and executed by per-resource schedulers, shown in (b).

The principles above represent the core ideas underlying monotasks. A variety of architectures could support these principles; for example, there are many approaches to scheduling monotasks. The remainder of this section focuses on the design decisions that we made to support monotasks specifically in the context of Apache Spark; we refer to the resulting system as MonoSpark.

5.3.2 How are multitasks decomposed into monotasks?

Decomposing jobs into monotasks does not require users to write their jobs as a sequence of monotasks, and can be done internally by the framework without changing the existing API. This is possible for frameworks such as Spark because they provide a high-level API: users describe the location of input data, the computation to perform on the data, and where to store output data, and Spark is responsible for implementing the necessary disk and network I/O. Because the framework is responsible for the details of how I/O occurs, it can replace the existing fine-grained pipelining with monotasks without requiring changes to user code.

With MonoSpark, the decomposition of jobs into monotasks is performed on worker

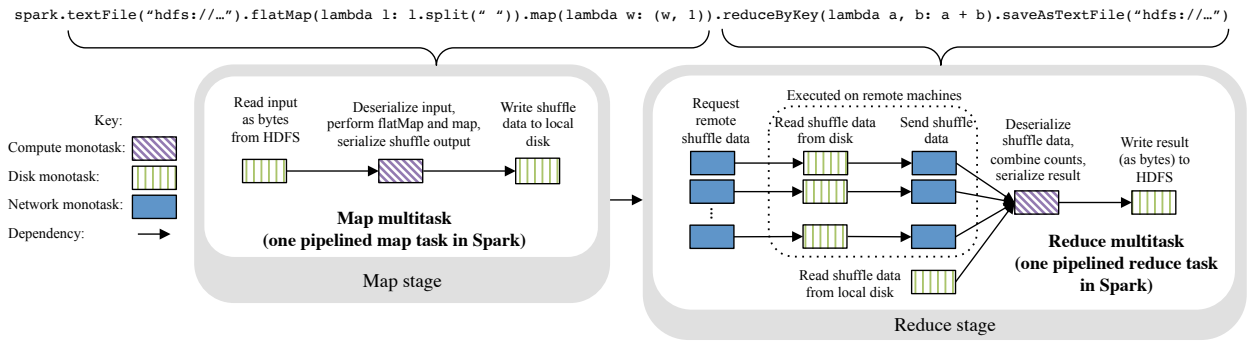


Figure 5.3: Decomposition of a Spark job that performs word count (written using Spark's Python API) into monotasks. The grey boxes show the two stages that the job would be broken into, and the white box within each stage illustrates how one multitask in the stage is decomposed into monotasks. For brevity the figure omits a compute monotask at the beginning of each multitask, which deserializes the task received from the scheduler and creates the DAG of monotasks, and a compute monotask at the end of each multitask, which serializes metrics about the taskâĂŹs execution.

machines rather than by the central job scheduler. The job scheduler works in the same way as in today's frameworks: it decomposes jobs into parallel multitasks that perform a computation on a single input data block, and that run on one machine. These multitasks are the same as tasks in current frameworks, and as in current frameworks, they are assigned to workers based on data locality: if a multitask reads input data that is stored on disk, it will be assigned to a worker machine that holds the input data.

Multitasks are decomposed into monotasks when they arrive on the worker machine. Figure 5.3 illustrates how jobs are decomposed into monotasks using the word count job from Figure 2.1 in Chapter 2 as an example. To read input data from HDFS, the job uses the `textFile` function, and passes in the name of a file in HDFS. With both Spark and MonoSpark, the job scheduler creates one multitask for each block of the file (HDFS breaks files into blocks, and distributes the blocks over a cluster of machines). Spark implements fine-grained pipelining to read and compute on the block: each Spark multitask pipelines reading the file block's data from disk with computation on the data. MonoSpark instead creates a disk read monotask, which reads all of the file block's bytes from disk into a serialized, in-memory buffer, followed by a compute monotask, which deserializes the bytes, emits a count of 1 for each instance of each word, and serializes the resulting data into a new in-memory buffer. The shuffle is similarly specified using a high-level API: the function `reduceByKey` is used to aggregate all of the counts for each word, which is done using a shuffle. Spark uses pipelining to parallelize reading shuffle data from disk and over the network with deserializing the data and performing the computation specified in the `reduceByKey` call. MonoSpark implements the shuffle by breaking each reduce multitask into network monotasks that each issue a request for shuffle data to remote machines. When remote machines receive the request, they create a disk read monotask to read all of the requested

shuffle data into memory, followed by a network monotask to send the data back to the machine where the reduce multitask is executing. Once all shuffle data has been received and placed in in-memory buffers, a compute monotask deserializes all of the shuffle data and performs the necessary computation. Other functions in Spark's API are implemented in a similar manner: Spark's API accepts a high-level description of the computation and the data to operate on, but not details of how resources are used, so MonoSpark can transparently change fine-grained pipelining to monotasks.

Because using monotasks eliminates the use of fine-grained pipelining to parallelize resource use, a single multitask will take longer to complete with monotasks than with today's frameworks. For example, for the map task in Figure 5.3, using monotasks serializes the resource use, causing the multitask to take longer than when resource use is parallelized using fine-grained pipelining (as in Figure 2.1). In order to avoid increasing job completion time, jobs must be broken down into enough multitasks to enable pipelining between *independent* monotasks in the same job. In other words, the monotasks design forgoes fine-grained pipelining within a single multitask and instead achieves high utilization through the statistical multiplexing of monotasks. We evaluate the sensitivity of MonoSpark's performance to the number of tasks in a job in §5.5.3, and discuss the resulting limitations on when MonoSpark can be used effectively in §5.8.

5.3.3 Scheduling monotasks on each worker

On each worker, monotasks are scheduled using two layers of schedulers. A top level scheduler, called the Local DAG Scheduler, manages the directed acyclic graph (DAG) of monotasks for each multitask. The Local DAG Scheduler tracks the dependencies for each monotask, and submits monotasks to the appropriate per-resource scheduler when all dependencies have completed. For example, for the map multitask in Figure 5.3, the Local DAG Scheduler will wait to submit the compute monotask to the compute scheduler until the disk read monotask has completed. The Local DAG Scheduler is necessary to ensure that monotasks can fully utilize the underlying resource and do not block on other monotasks during their execution.

The Local DAG Scheduler assigns monotasks to dedicated, per-resource schedulers that each seek to fully utilize the underlying resource while minimizing contention:

CPU scheduler The CPU scheduler is straightforward: one monotask can fully utilize one core, so the CPU scheduler runs one monotask per core and queues remaining monotasks.

Disk scheduler The hard disk scheduler runs one monotask per disk, because running multiple concurrent monotasks reduces throughput due to seek time.[1] Flash drives, on the other hand, can provide higher throughput when multiple operations are outstanding. The flash scheduler exposes a configuration parameter that allows users to change the number of concurrent flash monotasks based on the underlying hardware. For the flash drives we used,

[1]When disk monotasks transfer a small amount of data, disk throughput can be improved by running many parallel monotasks, so the disk scheduler can optimize seek time by re-ordering monotasks. We leave exploration of this to future work.

we found that using four outstanding monotasks achieved nearly the maximum throughput (results omitted for brevity).

Network scheduler Efficiently scheduling the network is the most challenging, because scheduling network monotasks requires coordination across machines. If a machine A starts sending data to machine B, the transfer may contend with other flows originating at A, or other flows destined to B. Determining how to match sender demand and receiver bandwidth to optimize utilization is an NP-hard problem that has been studied extensively. MonoSpark adopts a simple approach where all scheduling occurs at the receiver, which limits its outstanding requests. We chose the number of outstanding requests to balance two objectives. Consider a machine running multiple reduce multitasks that are each fetching shuffle data from all machines where map multitasks were executed. Issuing the requests for just one multitask at a time hurts utilization: the multitask could be waiting on data from just one (slow) remote machine, causing the receiving link to be underutilized. On the other hand, issuing the requests for too many multitasks at a time also hurts performance. Because the monotasks design relies on coarse-grained pipelining between monotasks for different multitasks, jobs complete most quickly when all of the data for one multitask is received before using any of the receiver bandwidth for the next multitask's data. This way the compute monotask (to process received data) can be pipelined with the next multitask's network requests. To balance these two issues, we limit the number of outstanding requests to those coming from four multitasks, based on an experimental parameter sweep. As in other parts of the monotasks design, more sophisticated schedulers (e.g., based on distributed matching between senders and receivers, as in pHost [33] and iSlip [46]) are possible, and we leave exploration of these to future work.

Queueing monotasks When more monotasks are waiting for a resource than can run concurrently, monotasks will be queued. The queuing algorithm is important to maintaining high resource utilization when multitasks include multiple monotasks that use the same resource. Consider multitasks that are made up of three monotasks: a disk read monotask, a compute monotask, and a disk write monotask. If a queue of disk write monotasks accumulates (e.g., when disk is the bottleneck), as writes finish and new multitasks are assigned to the machine, the read monotasks for the new multitasks will be stuck in the long disk queue behind all of the writes. Because each compute monotask depends on a read monotask completing first, the CPU will remain idle until all of the disk writes have finished. After a burst of CPU use, the disk queue will again build up with the disk writes, and future CPU use will be delayed until all of the writes complete. This cycle will continue and harms utilization because it prevents CPU and disk from being used concurrently. Intuitively, to maintain high utilization, the system needs to maintain a pipeline of monotasks to execute on all resources. To solve this problem, queues implement round robin over monotasks in different phases of the multitask DAG. For example, in the example above, the disk scheduler would implement round robin between disk read and disk write monotasks.

5.3.4 How many multitasks should be assigned concurrently to each machine?

The MonoSpark job scheduler works in the same way as the Spark job scheduler, with one exception: with MonoSpark, more multitasks need to be concurrently assigned to each machine to fully utilize the machine's resources. Consider Figure 5.2 as an example: with current frameworks, four tasks run concurrently (one on each core), whereas when jobs are decomposed into monotasks, four multitasks can concurrently be running CPU monotasks, while additional multitasks can use the disk and the network.

Before discussing how many multitasks to assign to each machine in MonoSpark, we note that the trade-offs in making this decision are different than in current frameworks. In current frameworks, the number of tasks concurrently assigned to each machine is intimately tied to resource use: schedulers must navigate a trade-off between assigning more tasks to each machine, to drive up utilization, and assigning fewer tasks, to avoid contention. With the per-resource monotask schedulers, on the other hand, each resource scheduler avoids contention by queuing monotasks beyond the number that can run efficiently on each resource. As a result, the job scheduler does not need to limit the number of outstanding tasks on each machine to avoid contention, and the primary risk is under-utilizing resources by assigning too few tasks per machine.

To ensure that all resources can be fully utilized, MonoSpark assigns enough multitasks that all resources can have the maximum allowed number of concurrent monotasks running, plus one additional monotask. For example, if a machine has four CPU cores and one hard disk, the job scheduler will assign ten concurrent multitasks: enough for four to have compute monotasks running, one to have a disk monotask running, four have monotasks running on the network, and one extra. The extra multitask exists for the purposes of the round-robin scheduling described in §5.3.3: without this extra monotask, one of the queues in the round-robin ordering can get skipped, because it is temporarily empty while a new multitask is being requested from the job scheduler.

The strategy emphasizes simplicity, and a more sophisticated scheduling strategy that accounted for the types of monotasks that make up each multitask and the current resource queue lengths on each worker machine would likely improve performance. The goal of this work is to explore the performance of a relatively simple and unoptimized monotasks design, so we leave exploration of more complicated scheduling techniques to future work.

5.3.5 How is memory access regulated?

Breaking jobs into monotasks means that more memory is used on each worker: in Figure 5.3, for example, all of the map multitask's data is read into memory before computation begins, whereas with current frameworks, fine-grained pipelining means that the data would be incrementally read, computed on, and written back to disk. This can result in additional garbage collection, which slows job completion time. This can also cause workers to run out of memory. Monotasks schedulers could prioritize monotasks based on the amount of

remaining memory; e.g., the disk scheduler could prioritize disk write monotasks over read monotasks when memory is contended, to clear data out of memory. We do not explore strategies to regulate memory use; we discuss this limitation in more detail in §5.8.

5.4 Implementation

The previous section described the design of MonoSpark; this section describes lower-level implementation details. MonoSpark uses monotasks to replace the task execution code in Apache Spark [3, 80].[2] MonoSpark is compatible with Spark's public API: if a developer has written an application on top of Spark, she can change to using MonoSpark simply by changing her build file to refer to MonoSpark rather than Spark. MonoSpark inherits most of the Spark code base, and the application code running on Spark and MonoSpark is identical. For example, for a job that filters a dataset from disk and saves the result, Spark and MonoSpark both read the same input data from the same place on disk, use the same code to deserialize the input data, run exactly the same Scala code to perform the filter, use the same code to serialize the output, and write identical output to disk. MonoSpark only changes the code that handles pipelining resources used by a task.

HDFS integration As part of integrating with Spark, our implementation also integrates with Hadoop Distributed File System (HDFS) [16], which is commonly used to store data that is analyzed with Spark. Spark uses a pipelined version of the HDFS API, where Spark reads one deserialized record at a time, and HDFS handles pipelining reading data from disk with decompressing and deserializing the data (writes work in a similar manner). In order to separate the disk and compute monotasks, we re-wrote the HDFS integration to decouple disk accesses from (de)serialization and (de)compression. We did this using existing HDFS APIs, so we did not need to modify HDFS.

Resource schedulers All of the per-resource schedulers are written at the application level and not within the operating system, meaning that resource use is not perfectly controlled. For example, we run one CPU monotask for each CPU core, but we do not pin these tasks to each core, and tasks may be interrupted by other monotasks that need small amounts of CPU (e.g., to initiate a disk read). We find that MonoSpark enables reasoning about performance in spite of these imperfections.

5.5 Monotasks performance

In this section, we compare performance of MonoSpark to Spark. We find that changing fine-grained pipelining to coarse-grained, single-resource monotasks does not sacrifice performance: MonoSpark provides job completion times within 9% of Apache Spark for typical scenarios.

[2]The MonoSpark code and scripts to run the experiments in the evaluation are available at https://github.com/NetSys/spark-monotasks.

5.5.1 Experimental setup

We ran all of our experiments on clusters of Amazon EC2 instances that have 8 vCPUs, approximately 60GB of memory, and two disks. Some experiments use m2.4xlarge instances with two hard disk drives (HDDs), while others uses i2.2xlarge instances with one or two solid state drives (SSDs), in order to illustrate that monotasks works with both types of disks. Our experiments compare Spark version 1.3 and MonoSpark, which is based on Spark 1.3. We ran at least three trials of each experiment, in addition to a warmup trial (to warmup the JVM) that was discarded, and except where otherwise noted, all plots show the median with error bars for the minimum and maximum values.

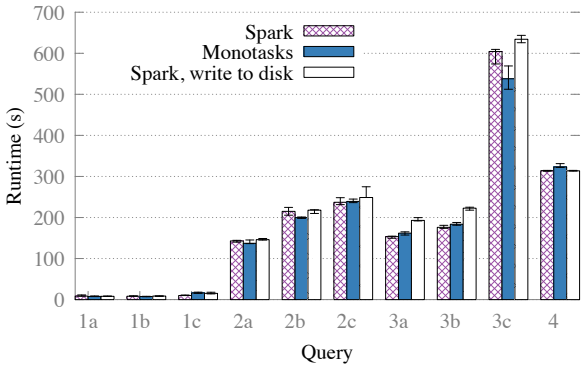
The version of Spark we compared against is known to have various CPU inefficiencies, and recent efforts have produced significant improvements in runtime, both in newer versions of Spark [76], and in alternative systems [27, 47]. As described in §5.4, we changed only the parts of Spark related to resource pipelining, so MonoSpark inherits Spark's CPU inefficiencies. MonoSpark would similarly inherit recent optimizations, which are orthogonal to the use of monotasks. For example, efforts to reduce serialization time would reduce the runtime for the compute monotasks that perform (de)serialization in MonoSpark. We also designed our workloads to avoid any single bottleneck resource, as we elaborate on in §5.5.2, and to include a workload (the machine learning workload) that avoids Spark-specific CPU overheads.

5.5.2 Does getting rid of fine-grained pipelining hurt performance?

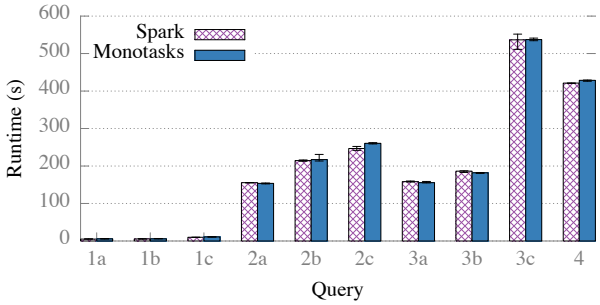
We evaluate MonoSpark using three workloads that represent a variety of applications and a variety of performance bottlenecks. In order to effectively evaluate whether changing fine-grained pipelining to single-resource monotasks harms performance, we chose workloads that do not have one resource with dramatically higher utilization than all other resources, as we elaborate on below. Workloads with several highly utilized resources put the most stress on strategies that re-arrange resource use, because resource use needs to be parallelized to maintain fast job completion time.

This section presents the high-level results of comparing MonoSpark to Spark for the three benchmark workloads; §5.5.3 and §5.5.4 describe differences in performance in more detail.

Sort The first workload sorts 600GB of random key-value pairs from disk using twenty worker machines that each have two hard disk drives. The job reads the input data from HDFS, sorts it based on the key, and stores the result back in HDFS. We tuned the workload to use CPU and disk roughly equally by adjusting the size of the value (smaller values result in more CPU time, as we elaborate on in §5.6.2). For this workload, Spark sorts the data in a total of 88 minutes (36 minutes for the map stage and 52 minutes for the reduce stage), and MonoSpark sorts the data in 57 minutes (22 minutes for the map stage and 35 minutes for the reduce stage). The reason MonoSpark is faster than Spark for this workload is discussed further in §5.5.4.



(a) 5 workers, each with 2 HDDs



(b) 5 workers, each with 2 SSDs

Figure 5.4: Comparison of Spark and MonoSpark for queries in the big data benchmark, using scale factor of 5, compressed sequence files, and 5 worker machines, running on two different cluster configurations. In (a), two configurations of Spark are shown: the default, and a configuration where Spark writes through to disk rather than leaving disk writes in the buffer cache.

Big Data Benchmark The big data benchmark [72] was developed to evaluate the differences between analytics frameworks and was derived from a benchmark developed by Pavlo et al. [64]. The input dataset consists of HTML documents from the Common Crawl document corpus [4] combined with SQL summary tables generated using Intel's Hadoop benchmark tool [79]. The benchmark consists of four queries including two exploratory SQL queries, one join query, and one page-rank-like query. The first three queries have three variants that each use the same input data size but have different result sizes to reflect a spectrum between business-intelligence-like queries (with result sizes that could fit in memory on a business intelligence tool) and ETL-like queries with large result sets that require many machines to store. The fourth query performs a transformation using a Python script. We use the same configuration that was used in published results [72]: we use a scale factor of five (which is the largest scale factor available) and a cluster of five worker machines that each have two HDDs.

Figures 5.4(a) compares runtime with MonoSpark to runtime with Spark for each query

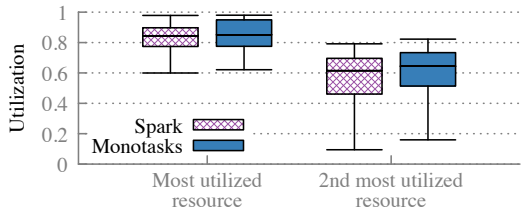


Figure 5.5: Utilization of the most utilized (i.e., bottleneck) resource, and the second most utilized resource during stages in the big data benchmark, for both Spark and MonoSpark. Boxes show the 25, 50, and 75th percentiles; whiskers show 5th and 95th percentiles.

in the workload. For all queries except 1c, MonoSpark is at most 5% slower and as much as 21% faster than Spark. Query 1c takes 55% longer with MonoSpark, which we discuss in more detail, along with the second configuration of Spark, in §5.5.3.

Figure 5.5 shows the resource utilization of the two most utilized resources on each executor during each stage of the big data benchmark queries, and illustrates two things. First, multiple resources were well-utilized during most stages. Second, MonoSpark utilized resources as well as or better than Spark.

We also ran the big data benchmark queries on machines with two SSDs (as opposed to two HDDs), to understand how different hardware affects the relative performance of Spark and MonoSpark. On SSDs, the MonoSpark is at most 1% slower than Spark and up to 24% faster, as shown in Figure 5.4(b).

Machine Learning The final workload is a machine learning workload that computes a least squares solution using a series of matrix multiplications[3] on a cluster of 15 machines, each with 2 SSDs. Each multiplication involves a matrix with one million rows and 4096 columns. Tasks in the workload each perform a matrix multiplication on a block of rows. This workload differs from the earlier workloads for three reasons. First, it has been optimized to use the CPU efficiently: matrices are represented as arrays of doubles that can be serialized quickly, and each task calls out of the JVM to optimized native code written using OpenBLAS [6]. Second, a large amount of data is sent over the network in between each stage; combined with the fact that CPU use has been optimized, this workload is network-intensive. Finally, the workload does not use disk, and stores shuffle data in-memory. As with other workloads, MonoSpark provides performance on-par with Spark, as shown in Figure 5.6.

5.5.3 When is MonoSpark slower than Spark?

MonoSpark can be slower than Spark in two cases. First, MonoSpark can be slower when a workload is not broken into sufficiently many multitasks. By using monotasks, MonoSpark eliminates fine-grained pipelining, a technique often considered central to perfor-

[3]The workload uses the block coordinate descent implementation in a distributed matrix library for Apache Spark, available at https://github.com/amplab/ml-matrix/.

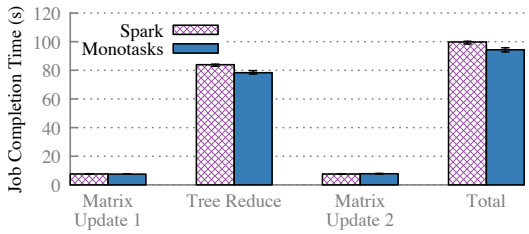


Figure 5.6: Comparison of Spark and MonoSpark for each stage of a machine learning workload that computes a least squares fit using 15 machines.

mance. MonoSpark's coarser-grained pipelining will sacrifice performance when the pipelining is too coarse, which occurs when the job has too few multitasks to pipeline one multitask's disk read, for example, with an earlier multitask's computation. Figure 5.7 plots the runtime of MonoSpark and Spark for a workload that reads input data from disk and then performs a computation over it, using different numbers of tasks. For each number of tasks, we repartition the input data into a number of partitions equal to the desired number of tasks. When the number of tasks is equal to the number of cores (the left most point), MonoSpark is slower than Spark, but as the number of tasks increases, MonoSpark can do as well as Spark by pipelining at the granularity of monotasks. This effect did not appear in any of the benchmark workloads that we ran because the default configuration for all of those workloads broke jobs into enough tasks to enable pipelining across different monotasks.

The second reason MonoSpark may be slower than Spark stems from how disk writes are treated: Spark writes data to buffer cache, and does not force data to disk. Disk monotasks, on the other hand, flush all writes to disk, to ensure that future disk monotasks get dedicated use of the disk, and because the ability to measure the disk write time is critical to performance clarity. This difference explains why query 1c in the big data benchmark was 55% slower with MonoSpark: each disk writes 511MB of data for MonoSpark but less than 200KB for Spark. We configured the operating system to force Spark to flush writes to disk and the resulting big data benchmark query runtimes are show in Figure 5.4(a). When Spark writes the same amount of output to disk as MonoSpark, query 1c is only 9% slower with MonoSpark.

5.5.4 When is MonoSpark faster than Spark?

In some cases, MonoSpark can provide faster runtimes than Spark. This is occurs for two reasons. First, per-resource schedulers control contention, which results in higher disk bandwidth for workloads that run on hard disk drives, due to avoiding unnecessary seeks. The effect explains MonoSpark's better performance on the sort workload and on some queries in the big data benchmark, where controlling disk contention resulted in roughly twice the disk throughput compared to when the queries were run using Spark.

Second, the per-resource schedulers allow monotasks to fully utilize the bottleneck resource without unnecessary contention. Figure 5.8 shows one example when MonoSpark achieved better utilization. The figure plots the utilization on one machine during a thirty

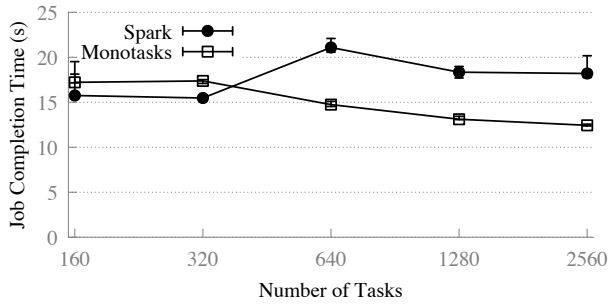


Figure 5.7: Comparison of runtime with Spark and MonoSpark for a job that reads input data and then computes on it, running on 20 workers (160 cores). Spark is faster than MonoSpark with only one or two waves of tasks, but by three waves, MonoSpark's pipelining across tasks has overcome the performance penalty of eliminating fine grained pipelining.

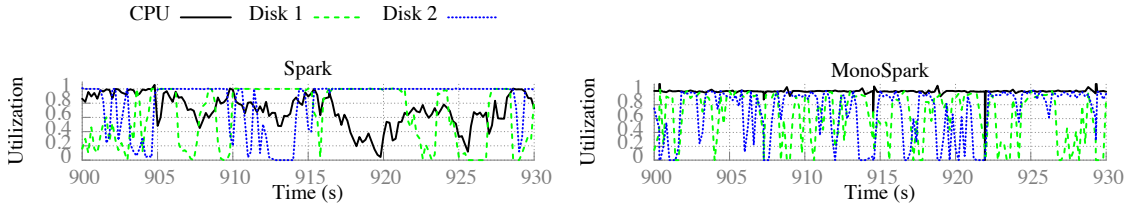


Figure 5.8: Utilization during the map stage of query 2c in the big data benchmark. With MonoSpark, per-resource schedulers keep the bottleneck resource fully utilized.

second period in the map stage of query 2c in the big data benchmark. Over the entire map stage, the per-resource schedulers in MonoSpark keep the bottleneck resource, CPU, fully utilized: the average utilization is over 92% for all machines. With Spark, on the other hand, each task independently determines when to use resources. As a result, at some points, tasks bottleneck on the disk while CPU cores are unused, leading to lower utilization of the CPU (75 - 83% across all machines) and, as a result, longer completion time. This problem would persist even with proposals that use a task resource profile to assign tasks to machines [37]. Resource profiles based on average task resource use do not account for how resource consumption arrives over the course of the task, so cannot avoid contention caused by fine-grained changes in resource use.

5.6 Reasoning about performance

Explicitly separating the use of different resources into monotasks allows each job to report the time spent using each resource. These times can be used to construct a simple model for the job's completion time, which can be used to answer what-if questions about

completion time under different hardware or software configurations. In this section, our goal is not to design a model that perfectly captures job completion time; instead, our goal is to design a model that is simple and easy to use, yet sufficiently accurate to provide estimates to users who are evaluating the benefit of different hardware or software configurations. We use the model to answer four "what-if" questions about how a job's runtime would change using a different hardware configuration, software configuration, or a combination of both. The model provides predictions within 28% of the actual runtime – even when the runtime changes by as much as a factor of 10. In addition, we show that our model makes it trivial to determine a job's bottleneck and demonstrate its use to replicate the bottleneck analysis results from Chapter 4. Finally, we evaluate the effectiveness of applying the same model to Spark.

5.6.1 Modeling performance

Decomposing jobs into monotasks leads to a simple model for job completion time. To model job completion time, each stage is modeled separately (since stages may have different bottlenecks), and the job completion time is the sum of the stages' completion times. Modeling the completion time of a stage involves two steps. First, information about the monotasks can be used to compute the ideal time spent running on each resource, which we call the ideal resource completion time. For the CPU, the ideal time is the sum of the time for all of the compute monotasks, divided by the number of cores in the cluster. For example, if a job's CPU monotasks took a total of twenty minutes, and the job ran on eighty cores, the ideal CPU time would be fifteen seconds, because all of the CPU monotasks could have completed in fifteen seconds if perfectly parallelized. For I/O resources, the ideal resource time can be calculated using the total data transmitted and the throughput of the resource:[4] ideal I/O resource time $= \frac{\text{sum of data transferred}}{\text{resource throughput}}$. For example, if a stage read twenty gigabytes from disk, and used ten disks that each provided 100 megabytes per second of read throughput, the ideal disk time would be twenty seconds.

The second step in building the model is to compute the ideal stage completion time, which is simply the maximum of any resource's completion time, as shown in Figure 5.9. In essence, the ideal stage completion time is the time spent on the bottleneck resource.

This model is simple and ignores many practicalities, including the fact that resource use cannot always be perfectly parallelized. For example, if one disk monotask reads much more data than the other disk monotasks, the disk that executes that monotask may be disproportionately highly loaded. Furthermore, all jobs have a ramp up period where only one resource is in use; e.g., while the first network monotasks are executing for a reduce task. Despite these omissions, we find that our model is sufficiently accurate to answer broad what-if questions, which we believe is preferable to a perfectly accurate but complex model that is difficult to understand and apply.

[4] For hard disk drives, because only one monotask is run concurrently, the ideal time can also be calculated by simply summing the monotask times and dividing by the number of disks in the cluster. This yields the same result as using the total data transmitted and the resource throughput.

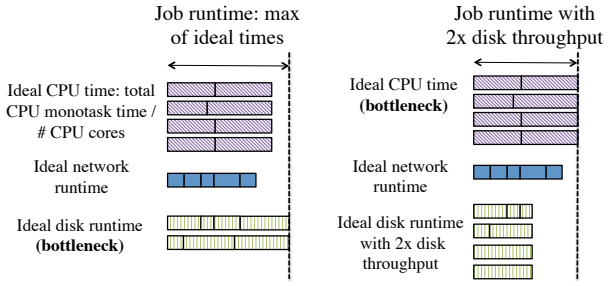


Figure 5.9: Monotask runtimes can be used to model job completion time as the maximum runtime on each resource. This example has 4 CPU cores and 2 hard disks.

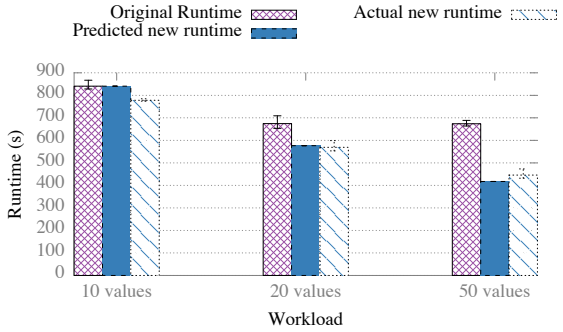


Figure 5.10: Monotask runtimes can be used to model job runtime on different cluster configurations. In this example, monotask runtimes from experiments on a cluster of 20 8-core, 1 SSD machines was used to predict how much faster the job would run on a cluster with twice as many disks on each machine.

The model for job completion time can trivially be used to determine the bottleneck, which is simply the resource with the longest ideal resource completion time.

5.6.2 Predicting runtime on different hardware

While understanding the bottleneck is useful, the model provides the most value in allowing users to answer what-if questions about performance. For example, to compute how much more quickly a job would complete if the job had twice as much disk throughput available, the ideal disk time would be divided by two, and the new modeled job completion time would be the new maximum time for any of the resources. An example of this process is shown on the right side of Figure 5.9. To compute the new estimated job completion time, we scale the job's original completion time by the predicted change in job completion time based on the model. This helps to correct for inaccuracies in the model; e.g., not modeling time when resource use cannot be perfectly parallelized.

Figure 5.10 illustrates the effectiveness of the model in predicting the runtime on a cluster with twice as many SSDs for three different versions of a sort workload. The figure shows

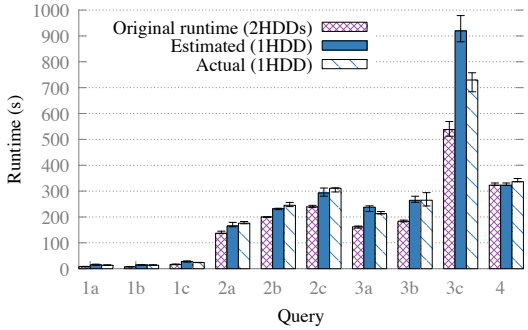


Figure 5.11: For all queries in the big data benchmark except 3c, the monotasks model correctly predicts the change from changing each machine to have only 1 disk instead of 2.

the runtime on a 20-machine cluster with one SSD per worker, the predicted time on a cluster with two SSDs per worker (based on monotask runtimes on the one SSD cluster), and the actual runtimes on a cluster with two SSDs per worker. The workload sorts 600GB of key-value pairs, where each value is an array of longs. Increasing the size of the value array while fixing the total data size decreases the CPU time (because fewer keys need to be sorted) while keeping the I/O demand fixed. We change the size of the value array (to be 10, 20, and 50) to evaluate how effective the model is for different balances of resources. With only 10 values associated with each key, the workload is CPU-bound, so the model predicts no change in the job's completion time as a result of adding another disk on each worker. In this case, the error is the largest (9%), because the workload does see a modest improvement from adding an extra disk (this improvement stems from reducing the duration of transient periods where the disk is the bottleneck, e.g., when starting a stage, where data must be read from disk before other resources can start). For the other two workloads, the model predicts the correct runtime within a 5% error. For both of these workloads, computing the new runtime with twice as many disks isn't as simple as dividing the old runtime by two. In at least one of the stages of both workloads, adding an extra disk shifts the bottleneck to a different resource (e.g., to the network) leading to a smaller than 2× reduction in job completion time. The monotasks model correctly captures this.

Figure 5.11 uses the model to predict the effect of removing one of the two disks on each machine in the big data benchmark workload from §5.5. A user running the workload might wonder if she could use just one disk per machine instead of two, because most queries in the workload are CPU bound (as discussed further in §5.6.5). The monotasks model correctly predicts that most queries change little from eliminating a disk: the predictions for all queries except query 3c are within 9% of the actual runtime.

The prediction has highest error for query 3c, when it overestimates the new runtime by 28%. This occurs because monotasks incorrectly predicts the change in runtime for a large shuffle stage that makes up approximately half of the job runtime. On-disk shuffles are the most difficult types of stage in which to get high utilization for both MonoSpark and Spark, because maintaining high utilization for the multitasks running on any one machine requires

a steady flow of shuffle data being read from *all* of the other machines in the cluster. A single slow disk or a machine that pauses due to garbage collection can block progress on all other machines while those machines wait for shuffle data from the slow machine (this is the same problem discussed for the 200-machine sort workload in §4.8). This problem is most pronounced when all three resources are used equally, in which case all resources on all machines must consistently be driven to high utilization to get the best performance. All three resources are used equally for the two-disk configuration, and performance is poor as a result: the average CPU, disk, and network utilization on each executor during this stage is roughly 50%. Since all resources are evenly bottlenecked, the model predicts that the runtime for that stage will increase by a factor of two once a disk is removed. Instead, when a disk is removed, MonoSpark is able to drive the bottleneck resource (now, the disk) to a higher utilization, and as a result, the stage only gets approximately 40% slower. Improving monotasks scheduling to enable uniformly higher resource utilizations would reduce this modeling error.

5.6.3 Predicting runtime with deserialized data

The model can also be used for more sophisticated what-if scenarios; e.g., to estimate the improvement in runtime if input data were stored in-memory and deserialized, rather than serialized on disk. This requires modeling two changes. First, data no longer will be read from disk. We account for this change by not including time for disk monotasks that read input data when we compute the ideal disk time (this is possible because each monotask reports metadata that includes whether the task was to read input or write output). Second, the job will spend less time using the CPU, because input data is already deserialized. MonoSpark separates the compute monotask into a first part that deserializes all of the data, and a second part that performs the remaining computation, and the compute monotask reports how long each part took. To model job completion time when data is already deserialized, we do not include the time to deserialize input data when modeling ideal CPU time. Using the new modeled ideal times, we calculate the new job completion time. We used this approach to predict the runtime of a job that sorted random on-disk data if data were stored deserialized in-memory, and the model predicted the new runtime within an error of 4% (the model predicted that the job's runtime would reduce from 48.5 seconds to 38.0 seconds, and the job's actual runtime with in-memory data was 36.7 seconds).

Separating the deserialization time from the rest of the computation is only possible because of the use of monotasks. Deserialization time cannot be measured in Spark because of record-level pipelining: Spark deserializes a single record and computes on that record before deserializing the next record, and processing a single record is too fast to time without significant overhead.

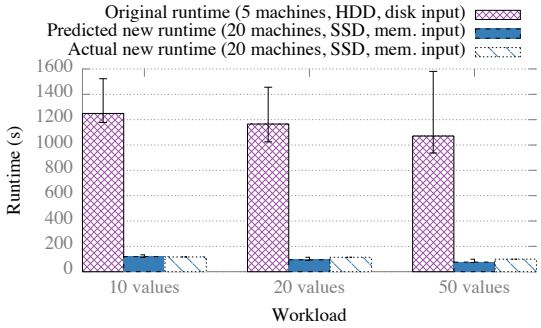


Figure 5.12: The monotasks model predicts the 10× improvement in runtime resulting from moving a workload from 5 machines with hard-disk-drives and input stored on-disk to 20 machines with solid-state-drives and input stored deserialized and in-memory with an error of at most 23%.

5.6.4 Predicting with both hardware and software changes

Thus far, we have shown how the monotasks model can be used to predict runtime changes resulting from a single change to the hardware or software configuration. The simple model can also accurately predict runtime changes stemming from multiple changes to both hardware and software. Figure 5.12 illustrates the change in runtime for 3 workloads that are moved from a 5-machine HDD cluster to a 20-machine SSD cluster. All workloads read 100GB of input data, sort it, and write it back to disk; the three workloads vary in the ratio of CPU to disk time, as in §5.6.2. The 100GB of input data did not fit in memory on the 5 machine cluster (it takes up approximately 200GB in memory), but with 20 machines, there is enough cluster memory to store input data in a deserialized format. In total, there are three changes associated with moving the workload to 20 machines: the workload runs on 4× as many machines (the number of tasks stays constant), input data is stored deserialized and in-memory rather than on-disk, and machines each have 2 SSDs rather than 2 HDDs, so shuffle and output data can be read and written more quickly. The monotasks model correctly predicts the resulting 10× change in runtime with an error of 23% in the worst case.

To give an example of the changes that occurred as a result of the hardware and software changes, the workload with 10 values was bottlenecked by disk on the 5-machine cluster in both the map and reduce stages. The monotasks model correctly predicted that, given the hardware changes, the map stage became CPU-bound (due to a combination of input data being stored in-memory, and the faster write time for the shuffle data), and the reduce stage became network bound (due to the faster read time for shuffle data and write time for output). One source of error for all three workloads was the network time. The model assumed that the same amount of data was sent over the network in both the 5-machine and 20-machine case. However, with 5 machines, each task could read 20% of its input data locally, whereas with 20-machines, an average of only 5% of input data could be read locally. As a result, the workloads on 20-machines sent more data over the network (and,

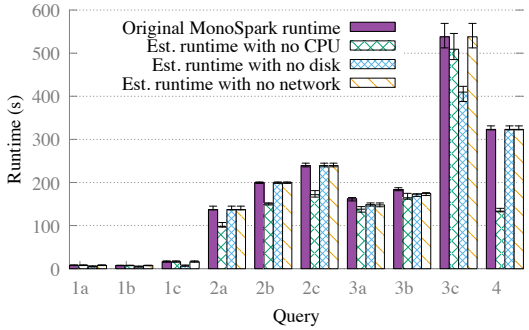


Figure 5.13: Monotask runtimes can be used to replicate previous work that used extensive logging to determine the job completion time if different resources were optimized to be infinitely fast (which serves as a bound on improvements from optimizing that resource).

correspondingly, spent more time using the network) than the model predicted.

5.6.5 Understanding bottlenecks

In addition to predicting runtime under different hardware and software configurations, our model can be used for simpler bottleneck analysis. Chapter 4 added instrumentation to Spark to measure times blocked on the network and disk, and used those measurements to determine the best-case improvement from optimizing disk and network: i.e., it used blocked times to determine how much faster jobs would run if they did not spend any time blocked on a particular resource. The analysis relied on adding extensive white-box logging to Spark; with monotasks, the necessary instrumentation (i.e., the runtime of different types of monotasks) is built into the framework's execution model. Typically the monotasks model predicts job completion time by taking the maximum of the ideal CPU, network, and disk times. The best-case job completion time if the disk were optimized can be computed by simply excluding the disk from the maximum, so instead taking the maximum over the ideal network and CPU times. Figure 5.13 illustrates these predictions using monotask runtimes for the big data benchmark used in Chapter 4. We arrive at the same findings as Chapter 4: for the big data benchmark, CPU is the bottleneck for most queries, improving disk speed could reduce runtime of some queries, and improving network speed has little effect. Some queries see improvement from optimizing multiple resources (e.g., query 3c) because the query consists of multiple stages that each have different bottlenecks.

5.6.6 Can this model be used for Spark?

This section explores whether the model we used to predict performance with MonoSpark could be applied to Spark, without re-architecting internals. Creating a model for job completion time is easy in MonoSpark because instrumentation for use of different resources is built into the framework architecture: resource use is separated into monotasks, and each

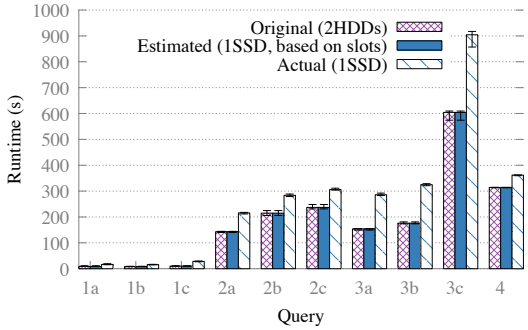


Figure 5.14: Using the number of slots to model the change in job completion time from reducing the number of disks does not result in correct predictions.

monotask reports how long it took to complete. While MonoSpark controls resource use with per-resource monotasks, Spark controls resource use with slots: as mentioned in §5.3.4, the job scheduler assigns tasks to machines in a fixed number of slots, and controlling this number of slots is the only mechanism the scheduler has for regulating resource use. The most straightforward way to apply the monotasks model to Spark is to use slots in the same way that the monotasks model uses monotasks: for example, if a job took 10 seconds to complete on a cluster with 8 slots, it should take 5 seconds to complete on a cluster with 16 slots. Figure 5.14 illustrates the effectiveness of this model, for the same prediction shown in Figure 5.11, where the runtime of the big data benchmark on machines with 2 HDDs is used to predict runtime on a cluster with 1 HDD per machine. Spark sets the number of slots to be equal to the number of CPU cores, so changing the number of disk drives does not change the number of slots. As a result, this model is inaccurate: it does not account for the slowdown that occurs when queries become disk bound. The user could scale the number of slots to 4 rather than 8, to account for the reduction in disk usage, but this would lead to predicting that queries would take twice as long with the reduced number of disks, which is only true for disk-bound queries. Fundamentally, the problem is that Spark uses one dimension, slots, to control resource use that is multi-dimensional.

Unfortunately, measuring Spark's resource usage to create a more sophisticated model is difficult. Spark tasks on a machine all run in a single process (the Java virtual machine), and resource usage of different concurrent tasks is interleaved by both Spark and the operating system scheduler. We measured this effect by running two different sort workloads concurrently: the 10-value and 50-value workloads from §5.6.2. We estimated resource use for each job by measuring the total resource use on each executor while each stage was running, and estimating each stage's resource use by scaling each executor's total resource use by the fraction of slots that were used by tasks for the stage. For example, if a stage ran for 10 seconds, and one executor with eight slots ran four, five-second tasks, we would divide the total resource use on the executor by four. Figure 5.15 shows this estimate, for the map stage in the 10-value job. These estimates are consistently incorrect, sometimes by a factor of two or more, because resource use is attributed equally to both jobs, rather than

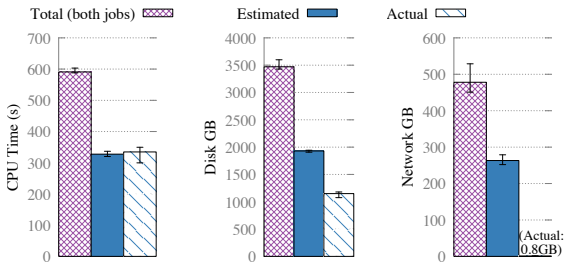


Figure 5.15: When multiple jobs are run concurrently in Spark, attributing resource to a particular job is difficult to do accurately. For one stage of one of the jobs, our estimate of resource usage was consistently inaccurate.

accounting for the jobs' different resource profiles. The median and 75th percentile error for all resources in both stages of both jobs is 17% and 68%, respectively, with Spark. Monotask times can easily be used to decouple resource use for the same two jobs: with MonoSpark, the error is consistently less than 1%.

Figure 5.15 illustrated why measuring resource use in Spark is difficult. If we *could* measure the resource use in Spark, would the model be accurate? While we cannot measure the resource use of each task, the monotask model relies on taking the aggregate resource use for an entire stage (by summing all of the monotask times for each resource, in each stage). We approximated this process in Spark by measuring the resource use on each executor while the big data benchmark is running in isolation. Because no other workloads were running, all of the resource use on each executor can be attributed to the stage. We used these resource uses as inputs to the monotasks model; Figure 5.16 illustrates the results. The model underestimates the increase in job completion time that results from using only one disk: the error ranges from 3%, for query 1a, to over 50%, for query 1c. This error stems from the challenges to reasoning about performance outlined in §5.2. In particular, one source of error is contention: when all tasks use one disk, contention leads to reduced disk throughput, which is not captured by the model.

We are able to model Spark performance only in a restricted case (when a job runs in isolation) and even in this case, the error was higher than the error for the same scenario using MonoSpark. This model can only be used for hardware changes; as mentioned in §5.6.3, it is not possible to measure deserialization time in Spark.

5.7 Leveraging clarity: auto-configuration

Because MonoSpark explicitly schedules the use of each resource, the framework has better visibility to automatically perform configuration that users are typically required to do. One configuration parameter that MonoSpark can set automatically is the appropriate amount of concurrency on each worker. Spark sets the number of concurrent tasks on each worker to be the number of cores on the worker, and allows users to change this by setting

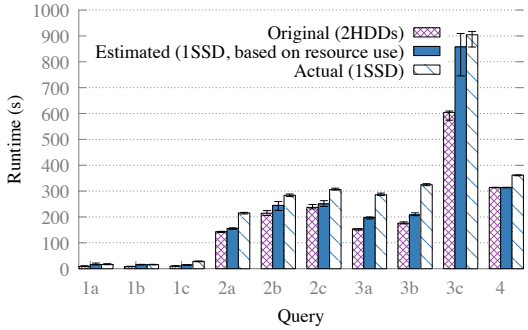
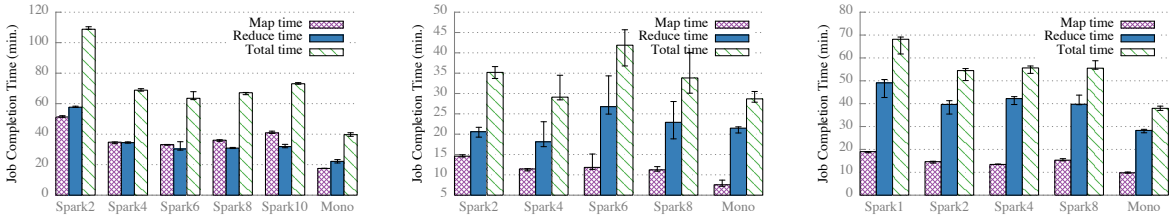


Figure 5.16: Even in cases where Spark's resource use can be measured to make a more accurate model than the slot-based approach in Figure 5.14, a Spark-based model has an error of 20-30% for most queries.



(a) Single Long (b) 25 Longs (c) 100 Longs

Figure 5.17: Runtimes for three different jobs, each under different configured numbers of tasks per machine with Spark (e.g., Spark2 is Spark with 2 tasks per machine). MonoSpark automatically configures the number of tasks per machine, and performs at least as well as the best Spark configuration for all three jobs.

a configuration parameter. Different configurations will work better for different workloads; for example, the user might want to increase concurrency to more than the number of cores if each worker machine has a large number of disks, and tasks are not CPU-intensive. MonoSpark eliminates this configuration parameter, because concurrency is controlled by each resource scheduler, as described in §5.3.4.

Figure 5.17 compares performance with MonoSpark to performance under a variety of Spark configurations, for three different workloads. The workloads each sort randomly generated key-value pairs by key, and differ in the number of values associated with each key, as in the workload in §5.6.2. The different characteristics of each workload mean that the best Spark configuration differs across workloads, and to extract optimal performance, a user would need to tune this configuration parameter to the correct value. MonoSpark automatically uses the ideal amount of concurrency for each resource, and as a result, performs at least as well as the best Spark configuration for all workloads. In some cases, MonoSpark performs as much as 30% better than Spark. This is for two reasons: first, Spark doesn't

allow users to change the number of concurrent tasks between the stages, and sometimes the ideal value differs for the two stages. Second, the disk monotask scheduler controls concurrent disk accesses to avoid unnecessary seeking.

5.8 Limitations and opportunities

This section begins with fundamental limitations of using monotasks, and then discusses opportunities to improve our current implementation.

Jobs with few tasks As illustrated in §5.5.3, if a job has a small number of multitasks relative to the available resources, then there may be no opportunities for coarse-grained pipelining across different monotasks. For the benchmark workloads we ran, we did not find this to be a problem, because the default configuration of all three workloads broke jobs into a sufficiently large number of multitasks. Many frameworks today encourage the use of a large number of small multitasks for performance reasons: having a large number of small tasks mitigates the effect of stragglers [61, 77] and helps to avoid situations where the task's intermediate data is so large that it needs to be spilled to disk [68]. Workloads that have only a single wave of multitasks will need to be broken into a larger number of smaller multitasks in order to run efficiently with monotasks, which can be done by changing a parameter when jobs are submitted.

Jobs with large tasks Frameworks like Spark and Hadoop allow tasks to handle more data than can fit in memory: if a task's intermediate data does not fit in memory, tasks will spill the data to disk and merge it at the end. However, in MonoSpark, data must fit in a buffer in-memory after it is read from (or before it is written to) persistent storage. As is the case for jobs with too few tasks, jobs with multitasks that are too large will need to be run with a larger number of smaller multitasks with monotasks.

Head of line blocking A monotask that reads a large amount of data from disk may block other tasks reading from that disk. This is not an issue with current frameworks because tasks share access to each resource at fine granularity. Using smaller tasks mitigates this problem with monotasks.

Memory use As mentioned in §5.3.5, MonoSpark uses more memory than Spark, because data is materialized in-memory between different types of monotasks. This memory pressure can alleviated by breaking jobs into a larger number of multitasks, which results in each multitask operating on less data, and, as a result, less data needing to be kept in-memory. The monotasks design could also be augmented to include a memory manager similar to the memory manager used in Themis [67].

Caching data Disk monotasks do not write data to the buffer cache because letting the OS manage the buffer cache hurts predictability. This hurts our performance compared to Spark for some jobs, as described in §5.5.3. MonoSpark could leverage Spark's application-level cache to opportunistically avoid writing data to disk.

Disk scheduling The disk monotask scheduler currently balances requests across available disks, independent of load. A better strategy would consider the load on each disk in

deciding which disk should write data; for example, writing to the disk with the shorter queue.

Multitask scheduling Our current implementation uses a simple multitask scheduler that assigns up to a maximum number of multitasks to each machine. This scheduler could be used to implement more sophisticated policies, e.g., to share machines between different users.

5.9 Related work

Performance clarity Most existing work approaches performance clarity by treating the system architecture as fixed, and either adding instrumentation or performing black-box experiments to reason about performance [7, 18, 40, 62]. Section 5.2 described existing approaches to performance clarity for data analytics frameworks. Ongoing debate about the bottleneck for data analytics frameworks suggests that reasoning about performance remains non-trivial [48, 49, 62, 71].

Performance clarity has been studied more extensively in the context of high-performance single-server applications. Causal profiling simulates the impact of performance optimizations by measuring the relative impact on performance of slowing down concurrent code [28]. If applied to large-scale distributed systems, causal profiling could answer the type of what-if questions that we used monotasks to answer. Flux takes an architectural approach, similar to monotasks: engineers use the Flux language to write high-performance servers, which enables both bottleneck identification and performance prediction [20].

Improving analytics system performance Numerous recent efforts have improved the performance of data analytics frameworks by optimizing CPU use; e.g., by reducing serialization cost, structuring computation to more efficiently use CPU caches, taking advantage of vectorization, and more [27, 48, 49, 76]. These efforts are orthogonal to monotasks: they make workloads less CPU-bound, but do not change the fine-grained pipelining of today's multi-resource tasks.

Improving resource scheduling A variety of scheduling projects have improved on slot-based scheduling models to account for use of multiple resources [34, 37]. These schedulers use estimates of task resource use to determine how many tasks to assign to each machine. They treat the structure of the task as a black box, and do not schedule a task's access to each resource. As a result, tasks may still contend, even if the average resource use of each task is less than the amount of resources available on the machine, as discussed in §5.5.4. Improving resource throughput with per-resource schedulers, as is done with monotasks, has been explored in various systems that use disk schedulers to batch access and avoid seeks (e.g., Themis [67] and Impala [25]).

Granularity of pipelining Traditional data processing systems use fine-grained pipelining to stream records between operators, as in the Volcano operator model [36]. Themis [67], for example, argues for record-at-a-time pipelining, where each record is processed fully after reading, to avoid memory pressure. We do not evaluate the very large scale (TBs of

data) workloads targeted by Themis. As discussed in §5.8, for such workloads, we anticipate borrowing their insight for reduced memory pressure: i.e., for such workloads, multi- and monotasks should act on fewer records.

Fine-grained pipelining has been revisited in the database literature to improve performance and take advantage of vectorized execution [63, 83, 44]. For example, SQL server 2012 abandoned the row-at-a-time iterator model, and instead processes a batch of (typically around 1000) rows at a time. Monotasks similarly abandons fine-grained pipelining, but with the different objective of easing users ability to reason about performance bottlenecks.

5.10 Conclusion

This chapter explored a new system architecture designed to provide performance clarity. We proposed decomposing today's multi-resource tasks into smaller units of work, monotasks, that each use only one of CPU, network, or disk. This decomposition trivially enabled the bottleneck identification that required using blocked time analysis in current systems. Using monotasks also allows for accurate estimates of the impact of potential system changes; these estimates enable users to determine which potential optimizations will yield the largest performance improvement for their workload. Using monotasks provides performance clarity without sacrificing high performance. Performance with our implementation is comparable to Apache Spark, and using monotasks presents new opportunities for optimizations that we have not yet fully explored. Since performance clarity can be provided without sacrificing high performance, we believe that designing for performance clarity should be a first-class concern in the architecture of future systems.

Chapter 6

Conclusion and future directions

This thesis proposed new architectures for data analytics frameworks that provide *performance clarity*: the ability to understand resource bottlenecks and to reason about the effect of potential hardware and software changes.

First, we proposed **blocked time analysis** to reason about performance bottlenecks in today's highly parallelized data analytics frameworks. Blocked time analysis uses extensive white-box logging to determine the importance of network and disk to a job's runtime.

We used blocked time analysis to study the performance of three data analytics workloads on Apache Spark. Our findings ran contrary to conventional wisdom about performance: we found that for the workloads we studied, jobs are often bottlenecked on CPU and not I/O, network performance has little impact on job completion time, and many straggler causes can be identified and fixed.

Based on the challenges we encountered in trying to achieve performance clarity through instrumentation alone, we explored a new architecture designed with the singular goal of providing performance clarity. We proposed decomposing today's multi-resource tasks into smaller units of work, **monotasks**, that each use only one of CPU, network, and disk. This decomposition simplified reasoning about performance and met both goals of performance clarity: it trivially enabled users to understand the system bottleneck, and also allowed for accurate estimates of the impact of potential system changes. Using monotasks brought performance clarity without sacrificing high performance. Performance with our simple implementation was comparable to Apache Spark, and using monotasks presents new opportunities for optimization that we have not yet fully explored.

6.1 Future directions

The techniques proposed in this thesis could be applied to understand performance in settings beyond data analytics frameworks. Blocked time analysis can be used more broadly to analyze the critical path in any parallel application, and to understand the impact of factors other than resource use on end-to-end performance. For example, loading a web page involves wide-area network communication to retrieve the page, server-side computation to

generate the page data, and client-side computation to render the page. These processes occur in parallel, and blocked time analysis could be used to estimate the potential impact of optimizing the wide-area network communication or the server-side computation of the page data. Using monotasks could similarly be done more broadly: we focused on using monotasks to replace multi-resource tasks in Spark, but monotasks could also be used in other data analytics frameworks (e.g., Naiad [50]) and for other types of distributed systems (e.g., key-value stores). Monotasks could be the basic building block for a new "datacenter operating system" that shares resources across different kinds of applications in units of monotasks. As datacenters become used for an increasingly large assortment of software, using monotasks as the basic building block across all applications could help ensure high resource utilization and allow finer grained sharing between diverse applications, all while maintaining the ability to reason about performance.

Looking beyond the techniques proposed in this thesis, we arrived at blocked time analysis and monotasks by focusing on performance clarity in a specific setting. We focused on data analytics frameworks, but many other existing systems would benefit from the ability to better reason about performance. We focused on resource use, and specifically on CPU, network, and disk, but many other factors of data analytics frameworks affect performance. Data analytics frameworks could benefit from improved performance clarity around memory use, garbage collection, how much time is spent on each operator in the computation graph (e.g., how much time is spent deserializing records, compared to applying a map function), and more. Finally, even in the context of resource use for data analytics frameworks, there likely exist many alternative architectures that, like monotasks, would provide performance clarity. We hope other system builders will propose new techniques and architectures focused on providing performance clarity.

6.2 Concluding thoughts

Some aspects of this thesis will inevitably have fading relevance; in particular, as mentioned in the conclusion to Chapter 4, our conclusions about the performance of today's workloads are already proving to be out of date as systems have evolved to address the performance bottlenecks described by our work.

While some methods and results in this thesis will become obsolete, we hope to leave readers with a belief in the importance of – and feasibility of – architecting for performance clarity. We view valuing performance clarity as taking a long-term view of how to optimize for the best performance. Without performance clarity, system designers will inevitably make mistakes in what they choose to optimize. Most engineers can probably recall a time when, faced with an important performance problem, engineers quickly began devising creative solutions – without first doing measurements to understand the source of the problem. This occurred for Hadoop when engineers implemented memory caching only to realize that serialization, and not disk I/O, was the bottleneck, so caching had little effect for existing map reduce workloads [74]. Sometimes engineers get lucky and these solutions work, but

they often lead to wasted engineering effort and unnecessarily complicate systems with optimizations that are not effective. Understanding a problem is typically the first step to solving it, and architecting systems to provide performance clarity will aid system designer in this crucial first step to effectively optimizing for high performance.

Performance clarity is also critical to enabling users to access the best performance. One example of the difficulty of configuring for the best performance – caused by a lack of performance clarity in current systems – can be seen in the common reaction to newly-published performance results. Often, someone will publish the results of comparing performance on numerous different systems, and the architects of systems that performed poorly will complain that if the publisher had better configured the system and consulted the architects for advice, performance would have been much better. However, the average user does not consult the system architect for advice on configuring the system, and the "naïve" published results are probably representative of the performance an average user would experience. The vast difference between performance of an average user's configuration and an expert's configuration illustrates that we have a long way to go in enabling users to access the best performance.

Providing and maintaining performance clarity will require a conscious effort by the community as a whole: obscuring performance factors sometimes seems like a necessary cost of implementing new and more complex optimizations, but inevitably makes understanding how to optimize performance in the future much more difficult. We have argued that obscuring performance factors need not be a cost of achieving high performance: that systems can be fast *and* easy to understand, so that as the system continues to evolve, developers and users alike continue to be able to optimize for the best performance.

Bibliography

[1] Apache Flink: Scalable Stream and Batch Data Processing. https://flink.apache.org/.

[2] Apache Parquet. http://parquet.incubator.apache.org/.

[3] Apache Spark: Lightning-Fast Cluster Computing. http://spark.apache.org/.

[4] Common Crawl. http://commoncrawl.org/.

[5] Databricks. http://databricks.com/.

[6] OpenBLAS: An optimized BLAS library. http://www.openblas.net/.

[7] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proc. SOSP*, 2003.

[8] M. Al-fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Proc. NSDI*, 2010.

[9] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *Proc. NSDI*, 2017.

[10] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: Coping with Skewed Content Popularity in MapReduce Clusters. In *Proc. EuroSys*, 2011.

[11] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective Straggler Mitigation: Attack of the Clones. In *Proc. NSDI*, 2013.

[12] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In *Proc. NSDI*, 2012.

[13] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu. GRASS: Trimming Stragglers in Approximation Analytics. In *Proc. NSDI*, 2014.

[14] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *Proc. OSDI*, 2010.

[15] G. Ananthanarayayan. Personal Communication, February 2015.

[16] Apache Software Foundation. Apache Hadoop. http://hadoop.apache.org/.

[17] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards Predictable Datacenter Networks. In *Proc. SIGCOMM*, 2011.

[18] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. In *Proc. SOSP*, 2004.

[19] D. Borthakur. Facebook has the world's largest Hadoop cluster! http://hadoopblog.blogspot.com/2010/05/facebook-has-worlds-largest-hadoop.html, May 2010.

[20] B. Burns, K. Grimaldi, A. Kostadinov, E. D. Berger, , and M. D. Corner. Flux: A Language for Programming High-Performance Servers. In *Proc. Usenix ATC*, 2006.

[21] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging Endpoint Flexibility in Data-intensive Clusters. In *Proc. SIGCOMM*, 2013.

[22] M. Chowdhury and I. Stoica. Coflow: A Networking Abstraction for Cluster Applications. In *Proc. HotNets*, 2012.

[23] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing Data Transfers in Computer Clusters with Orchestra. In *Proc. SIGCOMM*, 2011.

[24] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient Coflow Scheduling with Varys. In *Proc. SIGCOMM*, 2014.

[25] Cloudera. Cloudera Impala: Open Source, Interactive SQL for Hadoop. http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html.

[26] P. Costa, A. Donnelly, A. Rowstron, and G. O'Shea. Camdoop: Exploiting In-network Aggregation for Big Data Applications. In *Proc. NSDI*, 2012.

[27] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Çetintemel, and S. B. Zdonik. Tupleware: Redefining modern analytics. *CoRR*, 2014.

[28] C. Curtsinger and E. D. Berger. COZ: Finding Code that Counts with Causal Profiling. In *Proc. SOSP*, 2015.

[29] J. Dean. Personal Communication, February 2015.

[30] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. OSDI*, 2004.

[31] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *CACM*, 51(1):107–113, Jan. 2008.

[32] J. Erickson, M. Kornacker, and D. Kumar. New SQL Choices in the Apache Hadoop Ecosystem: Why Impala Continues to Lead. http://goo.gl/evDBfy, 2014.

[33] P. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker. pHost: Distributed Near-optimal Datacenter Transport Over Commodity Network Fabric. In *Proc. CoNext*, 2015.

[34] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proc. NSDI*, 2011.

[35] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proc. OSDI*, 2014.

[36] G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proc. SIGMOD*, 1990.

[37] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-Resource Packing for Cluster Schedulers. In *Proc. SIGCOMM*, 2014.

[38] B. Gufler, N. Augsten, A. Reiser, and A. Kemper. Load Balancing in MapReduce Based on Scalable Cardinality Estimates. In *Proc. ICDE*, pages 522–533, 2012.

[39] Z. Guo, X. Fan, R. Chen, J. Zhang, H. Zhou, S. McDirmid, C. Liu, W. Lin, J. Zhou, and L. Zhou. Spotting Code Optimizations in Data-Parallel Pipelines through PeriSCOPE. In *Proc. OSDI*, 2012.

[40] H. Herodotou. Hadoop performance models. *CoRR*, 2011.

[41] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs From Sequential Building Blocks. In *Proc. EuroSys*, 2007.

[42] V. Jeyakumar, M. Alizadeh, D. Mazieres, B. Prabhakar, C. Kim, and A. Greenberg. EyeQ: Practical Network Performance Isolation at the Edge. In *Proc. NSDI*, 2013.

[43] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. SkewTune: Mitigating Skew in MapReduce Applications. In *Proc. SIGMOD*, pages 25–36, 2012.

[44] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age. In *Proc. SIGMOD*, 2014.

[45] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *Proc. SoCC*, 2014.

[46] N. McKeown. The iSLIP Scheduling Algorithm for Input-queued Switches. *IEEE/ACM Trans. Netw.*, 7(2), 1999.

[47] F. McSherry, M. Isard, and D. G. Murray. Scalability! But at What Cost? In *Proc. Hot OS*, 2015.

[48] F. McSherry and M. Schwarzkopf. The impact of fast networks on graph analytics, part 1. http://tinyurl.com/qaw9lla, 2015.

[49] F. McSherry and M. Schwarzkopf. The impact of fast networks on graph analytics, part 2. http://tinyurl.com/q7aeajb, 2015.

[50] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A Timely Dataflow System. In *Proc. SOSP*, 2013.

[51] K. O'Dell. How-to: Select the Right Hardware for Your New Hadoop Cluster. http://goo.gl/INds4t, August 2013.

[52] Oracle. The Java HotSpot Performance Engine Architecture. http://www.oracle.com/technetwork/java/whitepaper-135217.html.

[53] K. Ousterhout. Display filesystem read statistics with each task. https://issues.apache.org/jira/browse/SPARK-1683.

[54] K. Ousterhout. Shuffle read bytes are reported incorrectly for stages with multiple shuffle dependencies. https://issues.apache.org/jira/browse/SPARK-2571.

[55] K. Ousterhout. Shuffle write time does not include time to open shuffle files. https://issues.apache.org/jira/browse/SPARK-3570.

[56] K. Ousterhout. Shuffle write time is incorrect for sort-based shuffle. https://issues.apache.org/jira/browse/SPARK-5762.

[57] K. Ousterhout. Spark big data benchmark and TPC-DS workload traces. http://eecs.berkeley.edu/~keo/traces.

[58] K. Ousterhout. Time to cleanup spilled shuffle files not included in shuffle write time. https://issues.apache.org/jira/browse/SPARK-5845.

[59] K. Ousterhout, C. Canel, S. Ratnasamy, and S. Shenker. Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks. In *Proc. SOSP*, 2017.

[60] K. Ousterhout, C. Canel, M. Wolffe, S. Ratnasamy, and S. Shenker. Performance Clarity As a First-class Design Principle. In *Proc. HotOS*, 2017.

[61] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica. The Case for Tiny Tasks in Compute Clusters. In *Proc. HotOS*, 2013.

[62] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making Sense of Performance in Data Analytics Frameworks. In *Proc. NSDI*, 2015.

[63] S. Padmanabhan, T. Malkemus, A. Jhingran, and R. Agarwal. Block oriented processing of Relational Database operations in modern Computer Architectures. In *Proc. ICDE*, 2001.

[64] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A Comparison of Approaches to Large-scale Data Analysis. In *Proc. SIGMOD*, 2009.

[65] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing The Network in Cloud Computing. In *Proc. SIGCOMM*, 2012.

[66] P. Prakash, A. Dixit, Y. C. Hu, and R. Kompella. The TCP Outcast Problem: Exposing Unfairness in Data Center Networks. In *Proc. NSDI*, 2012.

[67] A. Rasmussen, V. T. Lam, M. Conley, G. Porter, R. Kapoor, and A. Vahdat. Themis: An I/O-efficient MapReduce. In *Proc. SoCC*, 2012.

[68] S. Ryza. How-to: Tune Your Apache Spark Jobs (Part 2). goo.gl/7gjmyfcontent_copyCopyshortURL, 2015.

[69] K. Sakellis. Track local bytes read for shuffles - update UI. https://issues.apache.org/jira/browse/SPARK-5645.

[70] Transaction Processing Performance Council (TPC). TPC Benchmark DS Standard Specification. http://www.tpc.org/tpcds/spec/tpcds_1.1.0.pdf, 2012.

[71] A. Trivedi, P. Stuedi, J. Pfefferle, R. Stoica, B. Metzler, I. Koltsidas, and N. Ioannou. On The [Ir]relevance of Network Performance for Data Processing. In *HotCloud*, 2016.

[72] UC Berkeley AmpLab. Big Data Benchmark. https://amplab.cs.berkeley.edu/benchmark/, February 2014.

[73] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *NSDI*, 2016.

[74] A. Wang and C. McCabe. In-memory Caching in HDFS: Lower Latency, Same Great Taste. http://www.slideshare.net/Hadoop_Summit/inmemory-caching-in-hdfs-lower-latency-same-great-taste-33921794, 2014.

[75] D. Xie, N. Ding, Y. C. Hu, and R. Kompella. The Only Constant is Change: Incorporating Time-Varying Network Reservations in Data Centers. In *Proc. SIGCOMM*, 2012.

[76] R. Xin and J. Rosen. Project Tungsten: Bringing Spark Closer to Bare Metal. goo.gl/XBCbCycontent_copyCopyshortURL, 2015.

[77] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and Rich Analytics at Scale. In *Proc. SIGMOD*, 2013.

[78] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz. Wrangler: Predictable and Faster Jobs Using Fewer Resources. In *Proc. SoCC*, 2014.

[79] L. Yi, K. Wei, S. Huang, and J. Dai. Hadoop Benchmark Suite (HiBench). https://github.com/intel-hadoop/HiBench, 2012.

[80] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proc. NSDI*, 2012.

[81] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proc. OSDI*, 2008.

[82] J. Zhang, H. Zhou, R. Chen, X. Fan, Z. Guo, H. Lin, J. Y. Li, W. Lin, J. Zhou, and L. Zhou. Optimizing Data Shuffling in Data-Parallel Computation by Understanding User-Defined Functions. In *Proc. NSDI*, 2012.

[83] P. ÃĚke Larson, C. Clinciu, E. N. Hanson, A. Oks, S. L. Price, S. Rangarajan, A. Surna, and Q. Zhou. SQL Server Column Store Indexes. In *Proc. SIGMOD*, 2010.