Draft of Tuesday 8th August, 2017, 18:39.

Efficient and predictable data processing

Ionel Gog

University of Cambridge
Computer Laboratory
Corpus Christi College

January 2017

This dissertation is submitted for
the degree of Doctor of Philosophy

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except where specifically indicated in the text.

This dissertation is not substantially the same as any that I have submitted or that is being concurrently submitted for a degree, diploma, or other qualification at the University of Cambridge, or any other University or similar institution.

This dissertation does not exceed the regulation length of 60,000 words, including tables and footnotes.

Draft of Tuesday 8th August, 2017, 18:39.

Efficient and predictable data processing

Ionel Gog

Summary

Increasingly online computer applications rely on large-scale data analyses to offer personalised and improved products. These large-scale analyses are performed on *distributed data processing execution engines* that run on thousands of networked machines housed within an individual data center. These execution engines provide, to the programmer, the illusion of running data analysis workflows on a single machine, and offer programming interfaces that shield developers from the intricacies of implementing parallel, fault-tolerant computations.

Many such execution engines exist, but they often make assumptions about the computations they execute, or only target certain types of computation workflows. Understanding these assumptions involves substantial study and experimentation. Thus, developers find it difficult to determine which execution engine is "best", and even if they did, they become "locked in" to the engine of initial choice despite more appropriate engines becoming available. The range of execution engines also allows different workflow types to coexist within a data center, opening up both opportunities for beneficial sharing, and potential pitfalls for co-location interference. State-of-the-art data center schedulers struggle to make high quality decisions quickly in increasingly large data center clusters. Data center operators can use either schedulers that avoid co-location interference, but make decisions slowly, or schedulers that make decisions quickly, but with unpredictable computation runtime due to co-location interference.

In this dissertation, I first argue that the way we specify data computations should be decoupled from the execution engines that run the computations. I implement this decoupled design in my Musketeer proof-of-concept workflow manager. In Musketeer, developers express data computations using their preferred programming interface. These are then translated into a common intermediate representation from which Musketeer generates code and executes these computation on the most appropriate execution engine. I demonstrate that Musketeer reduces runtime of legacy data computations in a series of experiments on a medium-sized data center. Moreover, I show that Musketeer can be used to write data computations directly, because the code it automatically generates is competitive with optimised hand-written implementations.

Second, I present the min-cost flow-based Firmament scheduler that chooses good task placements. I describe the techniques I developed to improve Firmament to make decisions quickly in large data centers. I also introduce new scheduling features that were previously thought to be too expensive or impossible to offer using min-cost flow schedulers. I use one of these features in a case study, and I show that Firmament reduces network interference, and consequently workflow runtime. Finally, I demonstrate that Firmament chooses placements at least as good as other sophisticated schedulers, but at the speeds and scales associated with simple schedulers.

Acknowledgements

Contents

1 Introduction **16**

1.1 Contributions 17

1.2 Dissertation outline 19

1.3 Related publications 20

2 Background **23**

2.1 Cluster workloads 24

2.2 Data processing systems 27

2.3 Cluster scheduling 45

3 Musketeer: an extensible workflow manager **62**

3.1 Musketeer overview 65

3.2 Expressing workflows 68

3.3 Intermediate representation 74

3.4 Code generation 78

3.5 DAG partitioning and automatic mapping 84

3.6 Limitations and future work 91

3.7 Summary 93

4 Evaluation Musketeer **96**

4.1 Experimental setup and metrics 97

4.2 Overhead over hand-written, optimised workflows 98

4.3 Impact of Musketeer optimisations on makespan 102

4.4 Dynamic mapping to back-end execution engines 104

4.5 Combining back-end execution engines 107

4.6 Automated mapping . . . 108
4.7 Summary . . . 112

5 Firmament: a scalable, centralised scheduler **114**
5.1 Background . . . 115
5.2 Firmament overview . . . 120
5.3 Flowlessly: a fast min-cost flow solver . . . 122
5.4 Extensions to min-cost flow-based scheduling . . . 146
5.5 Network-aware scheduling policy . . . 156
5.6 Limitations . . . 157
5.7 Summary . . . 158

6 Evaluation Firmament **160**
6.1 Experimental setup . . . 160
6.2 Scalability . . . 162
6.3 Scheduling quality . . . 167
6.4 Summary . . . 170

7 Conclusions and future work **173**
7.1 Improving Musketeer . . . 174
7.2 Improving Firmament . . . 174
7.3 Summary . . . 174

Bibliography **175**

List of Figures

2.1 CDF of task runtime from a Google cluster trace . . . 25
2.2 CDF of task resources requests from a Google cluster trace . . . 25
2.3 Data center task life cycle . . . 26
2.4 Examples of front-end frameworks and back-end execution engines . . . 29
2.5 Examples of different dataflows models . . . 31
2.6 Examples of different graph processing models . . . 35
2.7 Makespan of a `PROJECT` and `JOIN` query in different data processing systems 41
2.8 Makespan of PageRank in different data processing systems . . . 43
2.9 Resource efficiency of different data processing systems . . . 44
2.10 Analysis of task CPU versus memory consumption from a Google cluster . . . 49
2.11 Resource requests normalised to usage from a Google cluster . . . 52
2.12 Comparison of different cluster scheduler architectures . . . 56

3.1 Coupling between front-end frameworks and back-end execution engines . . . 64
3.2 Decoupling between front-end frameworks and back-end execution engines . . . 64
3.3 Schematic of Musketeer's decoupling of front-ends from back-ends . . . 65
3.4 Phases of a Musketeer workflow execution . . . 66
3.5 PageRank workflow represented in Musketeer's IR . . . 77
3.6 Max-property-price workflow represented in Musketeer's IR . . . 81
3.7 Example of Musketeer's dynamic partitioning heuristic . . . 89
3.8 Example of a DAG optimisation inadvertently breaking operator merging . . . 92
3.9 Workflow on which Musketeer's dynamic heuristic misses a merge opportunity 93

4.1 Netflix movie recommendation workflow . . . 100
4.2 Musketeer-generated code vs. hand-written baselines on Netflix workflow . . . 101

4.3 Musketeer-generated code overhead for PageRank on the Twitter graph 101
4.4 Benefits of operator merging and type inference in Musketeer 103
4.5 Musketeer versus Hive and Lindi front-ends on TPC-H query 17 105
4.6 Musketeer's makespan on PageRank on different graphs 106
4.7 Musketeer's resource efficiency on PageRank on the Twitter graph 106
4.8 Comparison of back-end systems and Musketeer on cross-community PageRank 108
4.9 Musketeer DAG partitioning algorithms runtime 109
4.10 Makespan overhead of Musketeer's automated mapping choices 110
4.11 Makespan of SSSP and *k*-means on a 100 instances EC2 cluster. 111

5.1 Stages a task proceeds through in task-by-task queue-based schedulers 116
5.2 Stages min-cost flow-based schedulers proceed through. 116
5.3 Example of a simple flow network modelling a four-machine cluster 117
5.4 Example of a Quincy-style flow network 118
5.5 Quincy's scalability as cluster size grows 119
5.6 Architecture of the Firmament cluster scheduler 121
5.7 Examples of different types of flow network arcs 124
5.8 Runtime for min-cost flow algorithms on clusters of various sizes 132
5.9 Runtime for min-cost flow algorithms under high cluster utilisation 133
5.10 Example of a load-spreading flow network 133
5.11 Runtime for min-cost flow algorithms using a load-spreading policy 134
5.12 CDF of the number of scheduling events per time intervals in the Google trace . 134
5.13 Number of misplaced tasks when using approximate min-cost flow 136
5.14 Comparison of incremental and from-scratch cost scaling 139
5.15 Runtime reductions obtained by applying problem-specific heuristics 141
5.16 Schematic of Flowlessly's internals 143
5.17 Runtime reductions obtained by applying price refine before changing algorithm 144
5.18 Example of min-cost flow scheduler's limitation in handling data skews 148
5.19 Example showing how convex arc costs can be modelled in the flow-network . 149
5.20 Examples that create dependencies between tasks' flow supply 151
5.21 "and" flow network construct 152

5.22 Example of how gang scheduling can be represented in a flow-network. 153
5.23 Representation of gang scheduling tasks in the flow network 153
5.24 Token-based and worker-side weighted fair sharing represented in flow networks 154
5.25 Dominant Resource Fairness represented in the flow network 155
5.26 Example of a flow network for a network-aware scheduling policy 156

6.1 Task scheduling metrics 161
6.2 Comparison of Firmament's and Quincy's task placement delay 163
6.3 Firmament's algorithm runtime when cluster is oversubscribed 164
6.4 Firmament's scalability to short tasks 165
6.5 Firmament's latency on sped up Google trace 166
6.6 Percentage of tasks that achieve data locality with Firmament and Quincy . . . 168
6.7 Firmament's network-aware policy outperforms state-of-the-art schedulers . . . 169

List of Tables

2.1 Comparison of existing data processing systems 32
2.2 Comparison of existing cluster schedulers 55

3.1 Types of MapReduce jobs generated for Musketeer IR operators 80
3.2 Rate parameters used by Musketeer's cost function. 86

4.1 Evaluation cluster and machine specifications. 98
4.2 Modifications made to back-end execution engines deployed. 98

5.1 Worst-case time complexities of min-cost flow algorithms 131
5.2 Optimality requirements of different min-cost flow algorithms 138
5.3 Arc changes that require solution reoptimisation 140

6.1 Specifications of the machines in the local homogeneous cluster. 161

Listings

3.1 Hive query for the *max-property-price* workflow 70
3.2 BEER DSL code for the PageRank workflow 71
3.3 Musketeer's Lindi-compatible interface. 73
3.4 Gather-Apply-Scatter DSL code for PageRank. 74
3.5 Naïve Spark code for *max-property-price* 82
3.6 Optimized Spark code for *max-property-price* 83
3.7 Musketeer scheduling – high-level overview. 85
3.8 Dynamic programming heuristic for exploring partitionings of large workflows. 91
4.1 Hive code for *top-shopper* workflow. 102
5.1 Algorithm for extracting task placements from the flow returned by the solver. . 146

List of ToDos

1: Should I show the interface that must be implemented to integrate frameworks? . . 79

Chapter 1

Introduction

The growing desire for optimising products based on data-driven insights have lead to more and diverse data analysis workflows being executed in data center clusters. These workflows run on distributed data processing execution engines and conduct short interactive computations, graph analysis or batch data processing. The execution engines run computations on clusters of tens of thousands of commodity machines and offer easy-to-use programming interfaces that developers use to write workflows without worrying about how these are parallelised and on which cluster machines they are executed. Execution engines do not require workflow developers to write fault-tolerant code and instead provide interfaces that give the illusion of workflows running as a serial program.

Many distributed execution engines have been developed in recent years to target the aforementioned use cases. Each engine promises benefits over prior solutions, but they often make different assumptions, target different use cases and are evaluated under different conditions with varying workflows. For example, batch data execution engines optimise for processing huge data sets by parallelising workflows across many disks and machines. Graph processing systems, by contrast, spend significant time partitioning the graphs in order to reduce communication among machines. However, in practice, developers find it difficult to determine which system or combination of systems, is "best" for their workflows.

To make matters worse, developers often choose an initial execution engine and become "locked in". The introduction of a faster or more efficient execution engine does not currently guarantee rapid adoption because existing workflows must be manually ported – a task not undertaken lightly, even though rewriting may bring significant performance gains. Porting workflows between systems is tedious because user-facing *front-ends* that express workflows (e.g., Hive [TSJ+09], SparkSQL [AXL+15], Lindi [MMI+13]) are tightly coupled to *back-end execution engines* that run workflows (e.g., MapReduce [DG08], Spark [ZCD+12], Naiad [MMI+13], PowerGraph [GLG+12]). Developers implement workflows using front-end frameworks, but their workflows run only on the back-end execution engines that the front-ends are coupled to.

In this dissertation, I argue that the ways that workflows are defined should be decoupled from

the manner in which they are executed. This decoupling can be achieved by *(i) dynamically* mapping front-end workflow descriptions to a common intermediate representation; *(ii)* determining a good decomposition of workflows into jobs; and *(iii)* automatically generating efficient code for the chosen back-end execution engines out of a broad range of supported engines. To demonstrate this concept, I have built Musketeer, a workflow manager which breaks the tight coupling between front-end frameworks and back-end execution engines.

Once workflow expression and execution are decoupled, it is possible to choose the best combination of systems for a particular workflow in a data center in which workflow jobs are executed in isolation on homogeneous hardware. However, these jobs consist of many parallel *tasks* that run in data centers with heterogeneous hardware that is shared by many organisations, users, and distributed systems. Tasks' runtime and system-level performance vary significantly depending on the hardware they use and the interference caused by other tasks co-located on the hardware.

Data center cluster schedulers are responsible for placing tasks on machines in such a way that tasks do not interfere and run predictably. Cluster schedulers use elaborate algorithms to find high-quality task placements that take into account hardware heterogeneity and reduce task co-location interference [DK13; DK14; VPK+15]. However, these schedulers are *centralised* components that choose placements for entire clusters, and thus can take seconds or minutes to find placements [SKA+13; DSK15]. They fail to meet the scheduling latency requirements of interactive data processing jobs that must complete in a timely manner. As a result, clusters that run interactive jobs distribute placement decisions across several schedulers that use simple algorithms to place tasks with low scheduling delay [OWZ+13; RKK+16]. These *distributed schedulers* only have partial and often stale information about the cluster's state, and thus, they can choose poor task placements. Poor placements cause Musketeer-generated jobs and other jobs to interfere, and consequently experience performance degradations that make jobs unpredictable.

In this dissertation, I also argue that there is no need to make a trade-off between task placement quality and scheduling placement latency. In particular, I show that centralised cluster schedulers can both choose high-quality placements and offer low scheduling latency at scale. I extend the Firmament centralised scheduler, which is based on an expensive min-cost flow optimisation, with new key components that allow it to achieve low placement latencies. In a series of experiments, I demonstrate that Firmament quickly chooses high-quality placements. To achieve this, Firmament relies on several different optimisation algorithms, uses incremental algorithms where possible, and applies problem-specific optimisations.

1.1 Contributions

In this dissertation, I make three principal contributions:

1. My first contribution is a *model for decoupling data processing workflow specification from the manner in which workflows are executed*. I argue that workflows should be automatically translated into an intermediate representation, which can be optimised, and from which code can dynamically be generated at runtime for the best combination of execution engines. To explore the drawbacks and benefits of my model, I developed *Musketeer*, a workflow manager that dynamically translates user defined workflows to a range of data processing systems.

2. My second contribution is to show that *centralised data center schedulers scale to large clusters*. Centralised data center schedulers choose high-quality placements. However, this comes at the cost of high placement latency at scale which degrades runtime for interactive jobs and decreases data center utilisation. I extended the *Firmament* centralised scheduler to show that this perceived scalability limitation is not fundamental. Firmament uses the flow-based scheduling approach introduced by Quincy [IPC+09], which models scheduling as a minimum-cost flow optimisation over a graph, but which is known to take minutes to place tasks on large clusters. To address this limitation, I have developed *Flowlessly*, a minimum-cost flow solver that makes flow-based schedulers scale to tens of thousands of machines at sub-second placement latency in the common case. Flowlessly uses multiple min-cost flow algorithms, solves the problem incrementally when possible, and applies several problem-specific optimisations.

3. My third contribution is to *extend Firmament cluster manager* with a new scheduling policy, which uses one of the several scheduling features I have developed for flow-based schedulers (e.g., gang-scheduling fairness, resource hogging-avoidance, network-aware scheduling). These features were previously thought to be incompatible with flow-based schedulers.

1.1.1 Collaborations

The designs, architectures and algorithms presented in this dissertation are the result of my own work. However, colleagues in the Computer Laboratory have helped me implement several components that I later describe. In particular, Malte Schwarzkopf extended Musketeer with support for generating code for Metis jobs (§3.4). He has also implemented Firmament's components that spawn tasks, monitor and submit resource utilisation statistics from worker agents to the centralised coordinator (§5.2). Finally, Malte also implemented the load-spreading scheduling policy that I use to show the limitations of one of Flowlessly's min-cost flow algorithms (§5.3).

Natacha Crooks contributed to the implementation of Musketeer, adding, in particular support for a gather, apply, and scatter front-end framework (§3.2). Natacha extended Musketeer with traditional database query rewriting rules that optimise workflows in order to reduce their run-

time (§3.3.1). Moreover, she also extended Musketeer to integrate and generate job code for the Spark general-purpose data processing system (§3.4).

Matthew Grosvenor implemented the code that translates Hive workflows to Musketeer's intermediate workflow representation (§3.2). Finally, Adam Gleave contributed to Firmament by conducting an investigation of several minimum-cost flow algorithms in his Part II project under my co-supervision [Gle15].

1.2 Dissertation outline

This dissertation is structured as follows:

Chapter 2 gives an overview of the state-of-the-art data processing execution engines and identifies concepts shared by all of them. Moreover, in a series of experiments, I show that no execution engine always outperforms all others, and I highlight the challenges workflow developers are faced with in choosing between them. In this chapter, I also trace the recent developments in cluster scheduling, and discuss the requirements a scheduler must satisfy in order to place workflows such that they complete as soon as possible. In the discussion, I put an emphasis on the limitations of prior centralised and distributed schedulers.

Chapter 3 describes my model for decoupling data processing workflow specification from execution. I describe Musketeer, a proof-of-concept data processing workflow manager I built to showcase the model. Musketeer supports several front-end frameworks for developers to express their workflows, translates workflows into a common intermediate representation, and generates code to execute workflows in the best combination of data processing execution engines. First, I outline Musketeer's architecture, then discuss how workflows can be expressed when using it. Following, I discuss Musketeer's intermediate representation and how code for different frameworks is generated from it. Finally, I discuss how Musketeer decides on which combination of execution engines to run a workflow.

Chapter 4 investigates Musketeer's ability to efficiently run real-world data processing workflows. In a range of experiments, I show that Musketeer *(i)* generates efficient workflow code that achieves comparable performance to optimised hand-written implementations; *(ii)* speed-ups legacy workflows by mapping them to more efficient execution engines; *(iii)* flexibly combines several execution engines to run workflows; and *(iv)* automatically decides which engines are best to use for a given workflow.

Chapter 5 describes how centralised min-cost flow schedulers can be optimised to choose good task placements and scale to tens of thousands of machines at low scheduling latency. First, I explain how Firmament, an existing min-cost flow-based scheduler, works

and how it differs from traditional task-by-task schedulers. Following, I introduce Flowlessly, a min-cost flow solver I developed to make flow-based scheduling fast. Flowlessly automatically choses between different min-cost flow algorithms, solves the optimisation incrementally, and uses problem-specific heuristics. Finally, I describe how min-cost flow-based schedulers can be extended with features that were thought to be incompatible with such schedulers.

Chapter 6 evaluates Firmament's performance. First, I demonstrate in simulations using a Google workload trace from a 12,500-machine cluster that Firmament provides low scheduling latency even at scale. Second, I show that Firmament matches the scheduling latency of state-of-the-art distributed schedulers for workloads of short tasks, and exceeds their placement quality on a real-world cluster. Finally, I show that Firmament chooses better placements than state-of-the-art centralised and distributed schedulers.

Chapter 7 highlights directions for future work and concludes this dissertation. I consider challenges faced by Musketeer in supporting new data processing paradigms and systems (e.g., TensorFlow [MAP+15]) and discuss the extensions to Musketeer's intermediate representation required to support these systems. I also discuss how Firmament might be extended to consider additional types of resources when scheduling even more challenging future data center workloads.

1.3 Related publications

Parts of the work described in this dissertation are part of peer-reviewed publications:

[GSC+15] Ionel Gog, Malte Schwarzkopf, Natacha Crooks, Matthew P. Grosvenor, Allen Clement, and Steven Hand. "Musketeer: all for one, one for all in data processing systems". In: *Proceedings of the 10th ACM European Conference on Computer Systems (EuroSys)*. Bordeaux, France, Apr. 2015.

[GSG+16] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N.M̃. Watson, and Steven Hand. "Firmament: fast, centralized cluster scheduling at scale". In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Savannah, Georgia, USA, 2016, pp. 99–115.

I have also co-authored the following publications, which have influenced the work described in this dissertation, but did not directly contribute to:

[GSG+15] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. "Queues don't matter when you can JUMP them!" In: *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Oakland, California, USA, May 2015.

[GGS+15] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, et al. "Broom: sweeping out Garbage Collection from Big Data systems". In: *Proceedings of the 15th USENIX/SIGOPS Workshop on Hot Topics in Operating Systems (HotOS)*. Kartause Ittingen, Switzerland, May 2015.

[GIA17] Ionel Gog, Michael Isard, and Martín Abadi. *Falkirk: Rollback Recovery for Dataflow Systems*. In submission. 2017.

Chapter 2

Background

Many modern applications rely on large-scale data analytics workflows to provide high-quality services. These workflows run on hundreds of machines deployed in large data center clusters. The clusters comprise of tens of thousands of networked commodity machines, and are shared by multiple applications and users.

Developers of modern applications must solve two distinct but related problems in order to efficiently utilise data center clusters and to obtain good application performance:

Application implementation: implement applications and their associated data processing workflows such that: *(i)* they get high-performance, efficient data processing with minimal implementation effort; and *(ii)* applications remain compatible with future advances in parallel data processing.

Application execution: choose a set of machines on which to place applications. The selected machines must have sufficient available resources for the applications to run efficiently, and should meet other constraints such as tolerance of machine faults.

In this dissertation, I discuss two solutions that I propose to solve these problems: *(i)* a model for data processing that decouples data processing workflow specification from execution, and enables automatic and dynamic translation of workflows to the best combination of data processing systems; and *(ii)* a centralised cluster manager that chooses high-quality placements with low latency, and thus efficiently and predictably executes applications. Hence, in this chapter, I describe the properties and limitations of the state-of-the-art data processing systems and cluster managers that are used in today's data centers.

First, in Section 2.1, I briefly describe the types of workloads that are currently executed in data centers. These workloads have different resource requirements because they conduct various computations, and also vastly different scheduling latency expectations because their runtime can differ even by several orders of magnitude.

Following, in Section 2.2, I describe the different programming paradigms provided by the data processing systems available in today's data centers. I divide data processing systems into two categories: *(i)* front-end frameworks that are used by developers to express their workflows (§2.2.2), and *(ii)* back-end execution engines that run these workflows on hundreds of machines (§2.2.1). I look at how developers choice of front-end framework and back-end execution engine affects the efficiency of their workflows execution.

Finally, I focus on the challenges that arise from combining different types of workflows and applications in a single cluster. In Section 2.3, I outline the features a cluster scheduler must have in order to efficiently utilise resources, and to place workflows in such a way that they run predictably and complete as fast as possible. I also discuss the state-of-the-art cluster schedulers that aim to solve this problem today, and describe their limitations.

2.1 Cluster workloads

The workloads executed in data centers are becoming increasingly varied as more applications rely on distributed systems to service requests with low-latency and to provide insights obtained from large scale data analysis. The workloads consist of *jobs* that run many *tasks*. A task is an instance of an application executed as one or more processes in a container or virtual machine running on a single machine.

Many types of tasks with different runtimes and resource requirements execute in data centers. For example, in one of Google's clusters, the shortest 25% of tasks run for less than 180 seconds and the longest 25% of tasks run for more than 1,000 seconds (see Figure 2.1). Similarly, task resource requirements can vary greatly across tasks (see Figure 2.2). As a result, tasks can put conflicting demands on cluster schedulers. Short-running tasks need the schedulers to offer low placement latency in order to achieve low task runtime. By contrast, long-running tasks need schedulers to use expensive optimisations that choose high-quality placements which do not cause task runtime increases or performance degradation.

A common way to categorise data center tasks is by what kind of applications they run [SKA+13; VPK+15]. Tasks fall into one of the following three categories: *(i)* service tasks, *(ii)* batch tasks, or *(iii)* interactive tasks.

Service tasks are instances of high-priority applications or production-critical systems such as web servers, load balancers and databases. These systems must serve requests at all times, and thus service tasks run continuously until they are restarted due to hardware or software upgrades. In case of hardware failures, service tasks must be quickly migrated to other machines. Moreover, service tasks must run these user-facing systems predictably such that they meet service level latency agreements and maintain high serving rates. Cluster schedulers must place them on machines on which they do not interfere with other tasks and also make sure that no interfering tasks are later co-located.

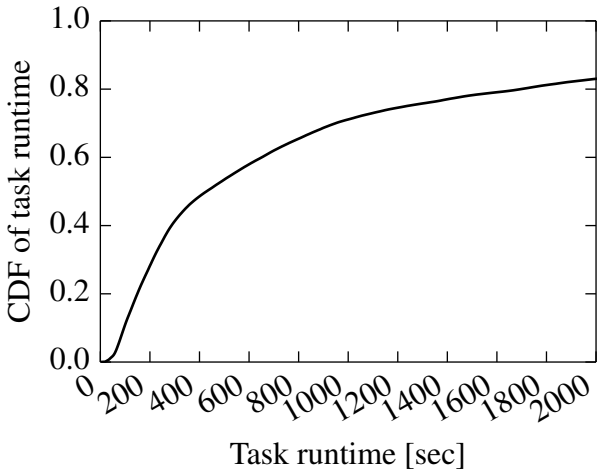


Figure 2.1: CDF of task runtime computed from a 30-day trace of a 12,500-machine Google cluster. Task runtimes vary greatly: 25% of tasks run for less than 180 seconds, and 75% of tasks run for less than 1,200 seconds. I do now show runtimes beyond 2,000 seconds because the trace contains service tasks that run until failure or for the entire trace duration – i.e., skew the CDF.

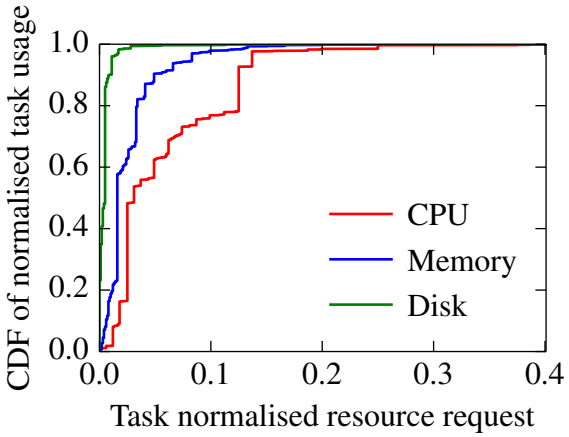


Figure 2.2: CDF of task resources requests from a Google cluster trace. Resource requests are normalised to the largest request. Task resource requests vary greatly; a small fraction of tasks requests few times more requests than other tasks.

Batch processing tasks are instances of infrastructure systems that run for limited time. Typical batch processing tasks run many types of offline computations such as extracting, transforming and loading (ETL) data. For example, Facebook's Bistro scheduler runs HBase compression batch tasks every 15 minutes [GSW15]. Moreover, batch processing tasks are widely used to run data analysis for improving product engagement and increasing revenues. In contrast to interactive tasks, batch tasks run computations whose failure does not cause application downtime. These offline computations do not have stringent placement requirements: tasks must not necessarily start simultaneously, computations are not sensitive to stragglers (i.e., tasks take longer to complete than other tasks) because they do not have to complete in real-time, and tasks do not have to be placed in different fault tolerance domains. Moreover, cluster schedulers can

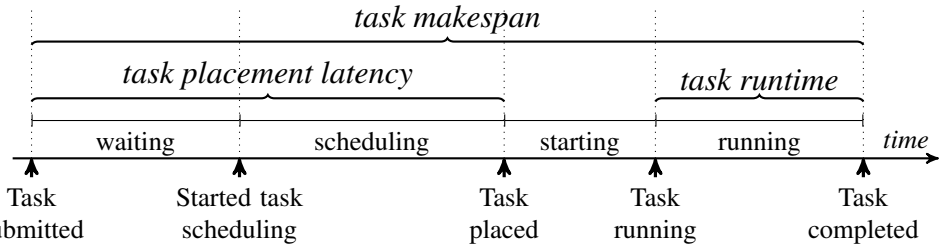


Figure 2.3: Data center task life cycle: state transitions events (bottom) and task-specific metrics (top).

preempt these tasks when other more important types of tasks must execute and no resources are available in the cluster.

Interactive data processing tasks execute simple queries submitted by data analysts, or dispatched by systems that provide personalised user responses. These tasks must complete as fast as possible (within seconds) because they are used in latency-sensitive systems such as online customer tools, monitoring systems and frameworks for interactive data exploration. These systems are becoming increasingly popular, and as a result more data center tasks are interactive. For example, 50% of the SCOPE data analytics tasks at Microsoft are interactive and run for less than 10 seconds [BEL+14, §2.3].

Interactive data processing tasks are challenging to schedule because: *(i)* they must be placed as fast as possible to avoid runtime increases, and *(ii)* many tasks must be scheduled simultaneously because otherwise some tasks may become stragglers, and significantly increase query completion time.

2.1.1 Task life cycle

All service, batch and interactive tasks go through the task life cycle that I show in Figure 2.3. Tasks are first submitted to a data center cluster manager, which schedules, runs and monitors tasks. After submission, tasks wait to be considered for placement by the cluster manager scheduler. Next, the cluster manager scheduler uses a specific algorithm to schedule tasks – i.e. chooses tasks placements. Upon task placement, the cluster manager downloads task binaries and setups containers or virtual machines for the tasks to execute in. Finally, tasks run until completion.

When I refer to the durations of the different stages of a task's life cycle, I use the following metrics (see Figure 2.3):

1. **Task placement latency** represents the time it takes the scheduler to decide where a task will run. I measure it from the time the task is submitted until the scheduler places the task on a machine.

2. **Task runtime** represents the time spent executing the user-provided computation in the task. I measure it from the time a task starts running until it completes. Task runtime does not include the time it takes to setup the task (e.g., setup dependencies, download binaries) [1].

3. **Task makespan** is an end-to-end metric that measures the entire time it takes to execute a task, from the moment the task is submitted until it completes [1].

Despite transitioning through same stages, tasks have different scheduling requirements. On one hand, service tasks run for a long time, and thus do not require quick placement, but it is important to choose good placements for them as they serve critical applications. On the other hand, interactive tasks must complete as soon as possible. They must be placed as fast as possible and transition through task stages quickly because otherwise their completion time might be significantly delayed.

In an ideal data center, tasks would share machines without affecting each others' performance; the cluster scheduler would choose placements instantaneously; and thus task makespan would equal task runtime. However, in the real world, cluster schedulers may cause unnecessary increases in task makespan because they may not choose high-quality task placements or may be unable to offer low task placement latency. Poor task placement quality decreases cluster utilisation, increases task makespan (for batch tasks), or decreases application-level performance (for service tasks). Similarly, high task placement latency increases task makespan, and decreases utilisation if tasks are waiting to be placed while resources are idle.

2.2 Data processing systems

As I noted in the previous section, data analytics jobs are increasingly diverse and have even more demanding compute and storage requirements. As a result, many systems for the parallel processing of big data have been developed in recent years (see Figure 2.4). These systems seek to give developers the ability to run parallel analyses on data sets whilst shielding them from the complexity introduced by data partitioning, communication, synchronisation and fault tolerance.

Many of these systems are designed to work well for specific workloads. For example, batch data processing execution engines process huge data sets spanning many disks in order to improve user engagement, or to extract, transform and load (ETL) data into other specialised systems. Real-time stream processing systems are used by applications for tasks such as detecting spam and drawing insights from data collected from devices in the Internet of Things. Graph processing systems execute jobs for detecting fraud and improving recommendation engines.

[1] Task runtime and makespan do not apply to service tasks because they do not complete.

Finally, interactive data analytics systems run tasks that complete within seconds and support user-facing services such as language translation and personalised search.

To guarantee good performance while maintaining simplicity, these systems are often specialised and embrace domain-specific optimisations. For example, systems for large-scale batch processing prioritise data throughput over quick recoveries from hardware failures [DG08]. Interactive data analytics systems use sampling or boundedly stale results [MGL+10], systems that run iterative workflows cache intermediate data in memory between iterations [ZCD+12; MMI+13], and graph processing systems adopt a vertex-centric synchronous computational model to provide simple programming interfaces without sacrificing performance [MAB+10; GLG+12; KBG12].

Additional specialised systems are being developed as new workflows and problem domains emerge. Today large organisations like Google, Facebook or Microsoft already run dozens of systems. In Figure 2.4, I show a subset of the data processing systems that have been developed over the past few years. Even though I only consider batch data and graph processing systems here, I notice that companies deploy more than a dozen different systems. As a result, developers of data processing workflow are faced with many choices of systems to use when implementing workflows.

These data processing systems can be categorised into two groups based on the type of abstractions they provide to developers:

Front-end frameworks are used by developers to express workflows using high-level declarative abstractions such as SQL-like querying languages or vertex-centric graph interfaces.

Back-end execution engines execute workflows expressed in front-end frameworks, or implemented using low-level abstractions (e.g., map and reduce functions, or stateful dataflow vertices).

Front-end frameworks offer interfaces that abstract away how and where workflows run. However, in practice many front-end frameworks are coupled to a single back-end execution engine; all front-end frameworks depicted in Figure 2.4 execute workflows on a single back-end. Developers chose a front-end framework to implement workflows, but as a result of this coupling, they do not benefit from the advantages different back-end execution engines have. Moreover, once they implement workflows, they find it tedious to port them to other frameworks. Developers become "locked in", despite faster or more efficient back-ends being available.

In the following subsections, I describe different dataflow models that the main batch and graph processing execution engines are based on (§2.2.1). I also explain how these models fundamentally limit back-end execution engines in certain situations. Following, I describe several state-of-the-art front-end frameworks (§2.2.2). Next, I show using several real-world experiments that choosing the "best" framework is difficult and that performance varies greatly depending on: *(i)* type of computation executed, *(ii)* input data size, and *(iii)* engineering decisions made

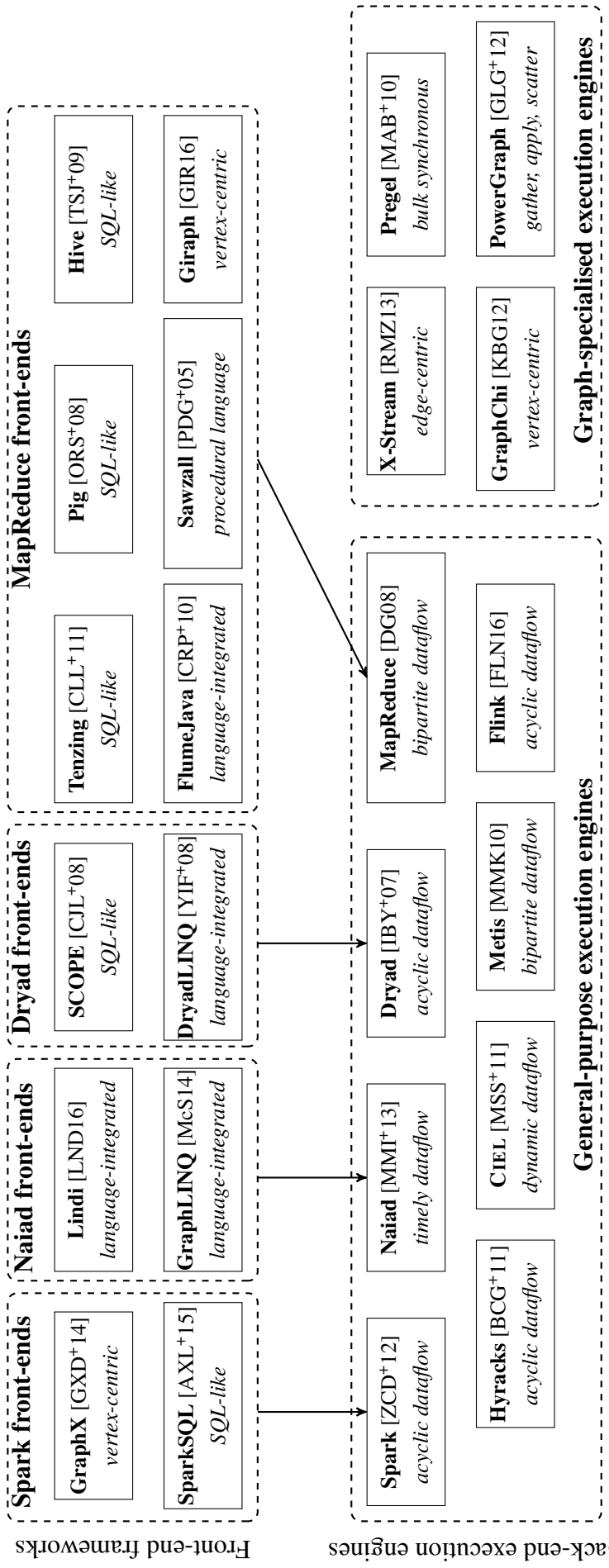


Figure 2.4: There are many front-end frameworks and back-end execution engines. Front-ends and back-ends typically have one-to-one mappings.

in the framework's development process (§2.2.3). Finally, I discuss several workflow managers – i.e., systems that create, optimise and dispatch data processing tasks. I focus on the properties of the currently available workflow managers and briefly touch upon the problems future workflow managers should address (§ 2.2.4).

2.2.1 Back-end execution engines

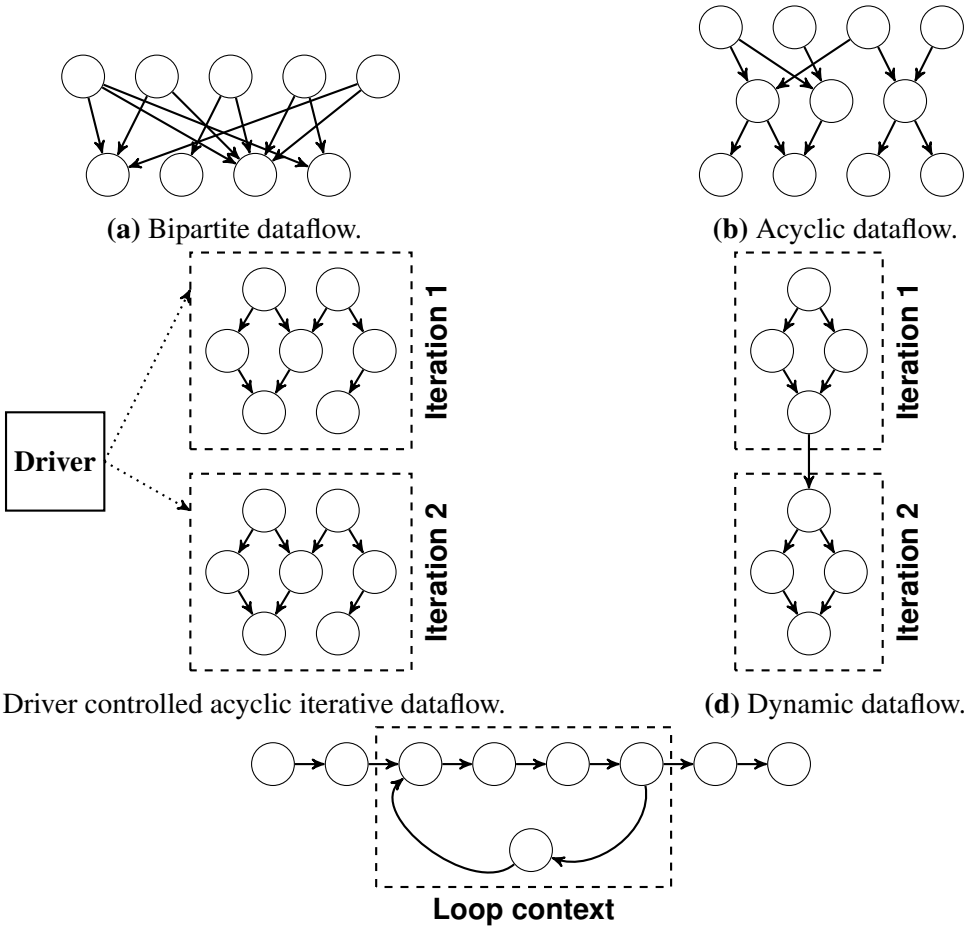
I turn now to describing the evolution of execution engines that run batch processing tasks and graph processing tasks. Purpose-built data processing engines exist for other types of tasks: for example, stream processing is conducted using Heron [KBF+15] and Storm [TTS+14] at Twitter, MillWheel [ABB+13] at Google, S4 at Yahoo! [NRN+10], Puma, Swift and Stylus at Facebook [CWI+16], and Samza at LinkedIn [SAM16]. However, in this dissertation I only focus on the evolution of batch and graph processing execution engines, but many of the concepts I introduce apply to stream processing systems as well.

2.2.1.1 Batch data processing back-end execution engines

Batch data processing systems are used to analyse "offline" large amounts of data in order to obtain data-driven insights. Data analysis was initially conducted on single commodity or mainframe machines. However, as data increased in size, disk I/O bandwidth became a bottleneck, or memory storage capacity limited how much data could be analysed. To address these limitations, analysis was parallelised over many disks and machines. This approach has become increasingly appealing as data centers transitioned from using high-performance computers (HPC) to using clusters of commodity machines. However, when using many commodity machines, failures occur on a regular basis [SPW09; GOF16]. Thus, the computation model used by distributed back-end execution engines must be robust to handle permanent or transient machine and network failures. The model must avoid fine-grained, costly coordination (e.g., no shared memory) or require determinism.

One model that meets these requirements is the parallel dataflow model. Initially proposed by Dennis as an alternative to the control flow architecture [Den68], the dataflow has been unsuccessful as a commercial computer architecture, but it inspired the dataflow programming model because of its intrinsic suitability for parallel computations.

The dataflow programming model defines computations as directed graphs. In these graphs, vertices conduct computations defined by users. The vertices apply pure functions (e.g., map, reduce) or complex, non-deterministic, stateful functions that may ingest data from external systems. Vertices are usually stateless, or if they are stateful then they only store data locally. In this model, a vertex's output is sent as input to the vertices it has outgoing arcs to. Thus, the dataflow programming model defines the flow of data explicitly by connecting graph vertices. The lack of shared vertex state and the explicit data coordination that is represented in the graph



(a) Bipartite dataflow.

(b) Acyclic dataflow.

(c) Driver controlled acyclic iterative dataflow.

(d) Dynamic dataflow.

(e) Timely dataflow.

Figure 2.5: Examples of the different dataflows models described in §2.2.1.1. Tasks are shown as circles and data flows along the arcs that connect them.

make dataflow workflows well-suited for running on large clusters of commodity machines in which failures are common.

I now describe the existing dataflow models and the back-end execution engines that are based on them (see Table 2.1 for a summary). In the prior literature, vertices and tasks are used interchangeably to refer to dataflow vertices. I call dataflow vertices "tasks" in order to distinguish between when I refer to them versus when I dicuss vertices in graph data.

Bipartite dataflow model. Back-end execution engines that are based on the bipartite dataflow model execute two types of tasks: *(i)* tasks that read input data and process it, and *(ii)* tasks that aggregate the processed data and output it (see Figure 2.5a). The programming interfaces exposed by these back-ends do not require developers to specify data dependencies among tasks because the back-ends create fixed dependencies among tasks.

Google's MapReduce [DG08] was perhaps the first big data execution engine and gained wide adoption (within Google). MapReduce influenced the design of other back-end execution engines (e.g., Apache Hadoop [HAD16], HaLoop [BHB+10]). MapReduce expects a developer

Data processing system	*Execution model*	*Environment*	*In-memory*	*Distributed I/O*	*Require pre-processing*	*Default sharding*	*Data sizes*	*Fault tolerance*	*Language*
Hadoop MapReduce [HAD16]	bipartite dataflow	cluster	–	✓	–	user-def.	large	✓	Java
HaLoop [BHB+10]	bipartite dataflow	cluster	✓	–	–	–	med.	–	C++
Metis [MMK10]	bipartite dataflow	machine	✓	–	–	user-def.	small	–	C++
Spark [ZCD+12]	acyclic dataflow	cluster	✓	✓	–	uniform	med.	✓	Scala
Dryad [IBY+07]	acyclic dataflow	cluster	–	✓	–	user-def.	large	✓	C#
Hyracks [BCG+11]	acyclic dataflow	cluster	–	✓	–	uniform	large	✓	Java
Flink [FLN16]	acyclic dataflow	cluster	✓	✓	–	uniform	med.	✓	Java/Scala
Ciel [MSS+11]	dynamic dataflow	cluster	(✓)	✓	–	user-def.	med.	✓	various
Naiad [MMI+13]	timely dataflow	cluster	✓	(✓)	–	user-def.	med.	(✓)	C#
Pregel [MAB+10]	vertex-centric	cluster	–	✓	–	uniform	med.	✓	C++
GraphChi [KBG12]	vertex-centric	machine	✓	–	✓	–	small	–	C++
Giraph [GIR16]	vertex-centric	cluster	–	✓	–	uniform	med.	✓	Java
PowerGraph [GLG+12]	gather, apply and scatter	cluster	✓	✓	✓	power-law	med.	(✓)	C++
X-Stream [RMZ13]	edge-centric	machine	✓	–	✓	–	med.	–	C++
Dremel [MGL+10]	execution trees	cluster	–	✓	–	uniform	large.	✓	C++

Table 2.1: A selection of contemporary back-end execution engines with their features and properties. (✓) indicates that the system can be extended to support this feature.

to express her workflow using two functions: *map* and *reduce*, and its back-end executes the workflow by first running *m* tasks that apply in parallel the user-provided map function over input data shards. The map function is applied in turn over each input data row and outputs zero or more key-value pairs. Next, MapReduce executes an intermediate step in which it sorts and groups by key the output from all *m* tasks. Subsequently, MapReduce sends shards of the grouped key-value pairs to *r* tasks that apply the user-provided reduce function. The function is applied on every key and list of grouped values, and outputs a key-value pair.

MapReduce execution engines are popular because they provide a simple programming model and they can tolerate hardware failures. They require developers to define only two functions, but the back-ends cannot execute complex workflows. For example, they restrict workflows to take a single input set and generate a single output set, and they cannot execute three-way joins on different columns in a single job. These limitations are addressed by high-level front-end frameworks (e.g., Pig [ORS+08], Hive [TSJ+09]) and workflow managers (e.g., Apache Oozie [OOZ16]). These systems provide interfaces to implement workflows that cannot be expressed in a single MapReduce job. For each workflow they generate and manage a direct acyclic graph (DAG) of MapReduce jobs. However, their approach suffers from several drawbacks: *(i)* workflows optimisations cannot be applied across job boundaries, *(ii)* resources are wasted while job output data are written to disk even if other jobs read it later, and *(iii)* additional systems must be managed.

Acyclic dataflow model. MapReduce execution engines cannot execute complex workflows because they are based on the bipartite dataflow model. The model encodes fixed dependencies between two types of tasks and cannot express computations that require more than two types of tasks. The more general acyclic dataflow model avoids this limitation: it uses a directed acyclic graph of tasks that are connected via directed edges if a task's output is another task's input (see Figure 2.5b). Tasks are not restricted to a single input and can be connected to any other tasks as long as no cycle exists in the graph. The acyclic dataflow model is suitable for parallel execution because all tasks that have input available can execute in parallel. Moreover, tasks that ingest other tasks' output can execute as soon as their dependencies start producing data.

Complex workflows can execute in a single job on back-ends based on the acyclic dataflow model, which avoids the need for separate workflow managers. For example, a workflow of MapReduce jobs can be modelled as a direct acyclic dataflow graph consisting of several connected bipartite dataflow graphs (i.e., MapReduce jobs). Moreover, Ke *et al.* show that complex workflows can be optimised more when they are executed in a single job [KIY13].

Dryad is the first general-purpose distributed back-end execution engine built using the acyclic graph dataflow model [IBY+07]. A Dryad job comprises of a DAG of vertex task nodes. Each task executes a user-specified computation. Tasks that are connected via graph edges have communications channels (e.g., TCP channels, shared memory) to send data between them. A

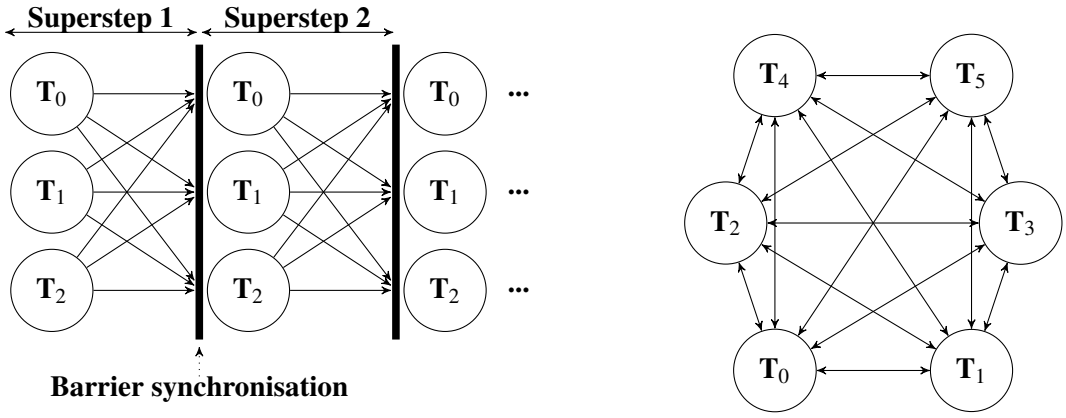
Dryad job completes when all tasks finish processing input or data sent by other tasks.

Like Dryad, Hyracks executes a DAG of user-specified task nodes [BCG+11], and Spark builds a DAG of "stages" that each correspond to several parallel tasks [ZCD+12]. Spark speeds up workflows by storing stores data in memory using resilient distributed data sets (RDD) rather than on disks [ZCD+12]. RDDs keep task output data in memory when possible and stream it to depedent tasks. By contrast, the bipartite execution engines write the output of every task to disk.

Back-end execution engines that use the acyclic graph dataflow model can run complex workflows, but there still are some workflows that they cannot natively execute in a single job. For example, they cannot execute workflows that contain data-dependent or unbounded iterations because their underlying dataflow model is acyclic. This limitation is addressed in back-ends using driver programs that execute each workflow iteration in a job, and check for computation completion. For example, the machine learning Apache Mahout library uses a driver program that submits cluster jobs that execute the iteration body, waits for them to complete, and finally checks if the iteration termination condition is met [MAH16]. Similarly, HaLoop is a modified version of the Hadoop MapReduce framework that internally verifies if the termination condition is met after each iteration [BHB+10]. Apache Flink [ABE+14; FLN16] uses a declarative fix-point operator that is repeatedly evaluated after each iteration. Spark and Dryad, by contrast, use extensions of high-level languages to support iterative workflows. DryadLINQ [YIF+08] is a front-end framework for Dryad in which iterative workflows are expressed by wrapping the directed acyclic graph in a C# for loop (§2.2.2). Similarly, developers can write iterative Spark workflows using Spark primitives (e.g., join, map, group by) and for loops provided by the high-level languages Spark supports (e.g., Scala, Python).

However, these solutions are not fault-tolerant; driver programs or DryadLINQ/Spark for loops execute on a single machine and maintain state (e.g., iteration count) vital for correct workflow completion. A failure of the machine on which they execute causes the entire workflow to restart from scratch.

Dynamic dataflow model. In order to address the limitations of the acyclic dataflow model, Murray *et al.* developed CIEL [MSS+11], which introduces a new, dynamic dataflow model, which natively supports workflows with data-dependent iterations. In the dynamic dataflow model, each task can spawn additional tasks at runtime (see Figure 2.5d). CIEL uses this feature to execute iterative workflows: for a given iterative workflow, CIEL first executes a set of tasks to conduct one iteration. When these tasks complete, one designated task checks if the iterative convergence criterion is satisfied, if not, it spawns a new set of tasks to conduct another iteration. This process repeats until the convergence criterion is satisfied. CIEL is a general back-end execution engine, but nested-loop workflows may be tedious to implement because developers must take care of the order in which tasks spawn additional tasks.



(a) BSP graph processing. Each circle represents a task/process that runs vertex-centric code for one or more vertices.

(b) GAS graph processing. Circles represent tasks that apply user-provided gather, apply or scatter functions. The tasks do not synchronise.

Figure 2.6: Examples of the different graph processing models described in §2.2.1.2.

Timely dataflow model. Timely dataflow was introduced by Murray *et al.*, and is a general model that does not expect developers to dynamically spawn tasks in order to implement iterative workflows. Timely dataflows are directed cyclic dataflows in which edges carry both data records and logical timestamps (see Figure 2.5e). These logical timestamps are used to pass iteration or workflow progress information among task nodes. The Naiad back-end execution engine is based on this timely dataflow model. Unlike many other systems that are not based on the timely dataflow model, Naiad is a general back-end that can execute a wide range of types of workflows: batch, iterative, incremental, streaming and graph workflows.

2.2.1.2 Graph data processing back-end execution engines

Graph-structured data is common in workflows that detect frauds, improve recommendation engines, or compute transportation routes. Initially, developers used general back-end execution engines that are based on models ill-suited to graph processing, such as bipartite or acyclic dataflow models [DG08; GIR16]. Many graph processing workflows run a series of iterations until a converge criterion is satisfied. The general back-ends perform poorly on graph workflows because they run at least a job per iteration, and do not optimise across iterations. Specialised graph processing back-ends addressed these limitations [MAB+10; GLG+12; KBG12; GIR16], and generally use one of two computations models that have limited expressivity, but can deliver significantly better performance on graph workflows.

Bulk synchronous parallel model. In 1990, Valiant introduced the bulk synchronous parallel (BSP) model for parallel computations with the purpose of bridging between software and hardware [Val90]. BSP algorithms run on a set of processes and proceed in a series of global supersteps. Each superstep comprises of three steps: *(i)* each process independently performs

local computation, *(ii)* processes exchange messages all-to-all, and *(iii)* processes wait (i.e., barrier synchronise) until all finish the previous two steps.

Pregel [MAB+10] was the first graph processing back-end to be based on the BSP computation model, and the first that provided a "vertex-centric" interface for developers to implement workflows. Developers provide code that Pregel executes in parallel for each graph vertex. In each superstep, the code receives data from adjacent vertices, runs the user-provided vertex computation, and finally sends data to adjacent vertices. Sent data are received by adjacent vertices in the following superstep (see Figure 2.6a). Pregel opened the way for other BSP-based systems such as Unicorn [CBB+13] and Apache Giraph [GIR16].

Despite their popularity, BSP-based graph processing back-ends are not suited for all graph workflows. McSherry *et al.* showed that on the graph connectivity problem, the single-threaded "Union-Find with weighted union" algorithm can outperform the equivalent distributed label propagation algorithm running on a specialised graph-processing back-end deployed on a 100-machine cluster by over an order of magnitude [MIM15]. However, frameworks that offer vertex-centric interfaces are not expressive enough to support the more efficient Union-Find algorithm.

Finally, BSP-based back-ends barrier synchronise processes before processing another superstep. Some processes may finish work early, and thus idle-wait for other processes to finish work and to dispatch their synchronisation message. To make matters worse, tasks may idle-wait for longer if the cluster network is under load and synchronisation messages are delayed – i.e., BSP-based back-ends are prone to be affected by other applications' network traffic. Together with my co-authors, we have shown that other applications' traffic can degrade the 99th synchronisation latency by 2-3× [GSG+15].

Asynchronous model. The process synchronisation step from the BSP model causes unnecessary slow-downs for some graph problems such as belief propagation and website ranking [LBG+12]. These algorithms work towards convergence criteria that can be achieved without synchronising processes at the end of each superstep. To the contrary, they converge faster if each process operates on the most recent data – even if the process has not yet received all messages from other processes.

Low *et al.* address these limitation in GraphLab, a graph processing back-end which supports asynchronous graph computations [LBG+12]. In GraphLab, user-defined vertex code can directly access its own state, adjacent vertices' state and adjacent edges' state. GraphLab runs the vertex code in parallel, but also ensures that neighbouring vertices do not run at the same time so that it keeps state consistent.

PowerGraph introduces the three-phase **G**ather, **A**pply, **S**catter (GAS) abstraction which can express both synchronous and asynchronous graph computations [GLG+12]. In the gather phase, data are collected from adjacent vertices and edges using a user-defined commutative and associative *gather* function. The function's output is then passed to the *apply* user-defined function

that updates the vertex state. Finally, the *scatter* filters the vertex state and sends it to adjacent vertices.

Both synchronous and asynchronous graph processing back-ends must decide where vertex code is executed. This is a key problem they must solve because most graph workflows do little work per graph vertex, and as a result, back-ends spend most of workflow runtime synchronising and sending data among processes. Communication and implicitly workflow runtime can be significantly reduced if input graphs are partitioned across machines such that adjacent vertices are co-located on the same machine. However, in practice most frameworks use simple graph partitioning algorithms that do not co-locate many neighbouring vertices, or use the a single partitioning algorithm for all types of input graphs.

Optimising graph processing frameworks. Gonzales *et al.* observe that many social and web graphs have power-law degree distributions, which are difficult to partition [GLG+12]. They address this problem with a new vertex-cut graph partitioning algorithm that reduces communication. Chen *et al.*, improve upon PowerGraph's performance with PowerLyra, a back-end that dynamically applies different variants of a partitioning algorithm depending on the graph's structure [CSC+15]. PowerLyra utilises a partitioning algorithm that combines PowerGraph's vertex-cut and edge-cut with few heuristics. Zhang *et al.*, improve further on PowerLyra's performance with a 3D partitioning algorithm that reduces network communication for matrix-based applications that are executed as graph workflows [ZWC+16].

Others take optimise graph processing back-ends for single machine execution. For example, GraphChi [KBG12] leverages SSDs for resource-efficient vertex-based graph workflows on a single machine, while X-Stream [RMZ13] uses an edge-centric approach that is optimised to sequentially read input graphs, also on a single machine.

2.2.2 Front-end frameworks

Many big data workflow developers are data analysts who find it difficult to implement workflows using low-level interfaces provided by back-end execution engines. Front-end frameworks support higher-level abstractions that make it easy to express workflows. These frameworks translate and execute workflows into jobs, which run on back-end execution engines.

In practice, front-end frameworks are almost always coupled to a single back-end execution engine. They cannot execute workflows on other back-ends (see Figure 2.4 for examples of coupling between front-end frameworks and back-end execution engines). This hinders the adoption of new more efficient back-ends: legacy workflows must be manually ported to the appropriae back-end. Porting is tedious and is not undertaken lightly, especially in large companies where hundreds of thousands of workflows would have to be ported manually.

In this dissertation, I focus on front-end frameworks for batch and graph data processing. In addition, there are many front-end frameworks for representing stream and interactive data work-

flows (e.g., PowerDrill [HBB+12], Peregrine [MG12]). Many of the techniques I describe can be applied to these as well.

2.2.2.1 Batch data processing front-end frameworks

Batch back-end execution engines processes huge data sets quickly because they parallelise I/O and processing over many disks and machines. It is paramount for their front-end frameworks to provide intuitive ways of expressing workflows without restricting parallelism.

Structured Query Language (SQL). SQL is a declarative language that is intuitive and is extensively used as an interface for interacting with databases. SQL comprises of a small set of operators that are based on Codd's relational algebra (e.g., filter, project, join) [Cod70]. These operators are data-parallel because each operator can be simultaneously applied on partitions of the data. For each SQL query, database parsers create a logical query plan, which stores each query as a directed acyclic graph of operators in which arcs connect operators that have a data dependency between them. Substantial research effort has gone into algorithms that reorder the operators in the logical query DAG in such a way that the query semantics are the same, but the amount of computation required to produce the output data are reduced. For example, many algorithms push filter operators as early as possible in the DAG in order to reduce the amount of data processed by depedent operators.

SQL's widespread use, the logical query plan DAG representation that maps to an acyclic dataflow workflow, and the abundance of query plan optimisation algorithms make SQL to be an attractive way of expressing workflows. For example, Sawzall [PDG+05] introduces a new procedural programming language in which big data workflows are expressed in two steps: *(i)* a step in which the program only performs operations on a single data record at a time, and *(ii)* a step in which the output from the first step is processed using data aggregators. The language is less sophisticated than SQL, but it is easily translatable to MapReduce jobs. Pig [ORS+08] and Hive [TSJ+09] are widely used front-end frameworks and workflow managers that present a SQL-like interface to developers. They both run on top of the Hadoop MapReduce back-end and translate SQL-like workflows to directed acyclic graphs of MapReduce jobs. They retrofit support for acyclic complex workflows on top of MapReduce's limited bipartite dataflow model: workflows execute as a series of MapReduce jobs in topological order of their data dependencies. Shark [XRZ+13] replaces Hive's physical query plan generator to use Spark's resilient distributed data sets rather than generate several MapReduce jobs, and thus supports fast interactive in-memory queries. Spark SQL [AXL+15] is a redevelopment of Shark that offers better integration between relational SQL code and traditional Spark procedural processing code. SCOPE [CJL+08] and Tenzing [CLL+11] make the relationship to SQL more explicit; Tenzing provides an almost complete SQL implementation on top of MapReduce. The semantics of these frameworks, however, are heavily influenced by back-end execution engines to which

they translate workflows. For example, Pig relies on `COGROUP` clauses to delineate MapReduce jobs, while Spark SQL is tightly integrated with a Spark-centric query plan optimiser.

Language-integrated solutions. Often, data processing workflows are deeply integrated in applications. In such cases, applications and workflows can be developed using languages for which integrated workflow front-ends exist. For example, FlumeJava is a Java library that provides few immutable parallel collections which support several operations for parallel processing (e.g., ParallelDo, GroupByKey) [CRP+10]. Instead of eagerly executing these operations, FlumeJava defers their evaluation and constructs an acyclic dataflow graph. Following, it optimises the dataflow graph, and it executes the workflow on either the local machine or a MapReduce cluster depending on the input data size.

Similarly, **L**anguage **IN**tegrated **Q**uery (LINQ) is a .NET framework that adds SQL-style query extensions as syntax sugar to the programming languages running on top of .NET. LINQ abstracts away workflow definition from execution by supporting several "query providers" (e.g., provider to XML, provider to SQL). Yu *et al.* developed DryadLINQ, a new LINQ query provider that translates LINQ expressions and executes them in parallel on a cluster using the Dryad back-end execution engine [YIF+08]. Similarly, Lindi is a query provider that executes workflows on the Naiad back-end execution engine [LND16].

2.2.2.2 Graph data processing front-end frameworks

Some specialised front-end frameworks offer interfaces that are well-suited for expressing graph processing workflows. Pregelix [BBJ+14] is a front-end framework that provides the same vertex-centric interface as Pregel, but internally models the graph computation as a traditional database query of relational operators. By taking this approach, Pregelix is able to execute workflows on the Hyracks [BCG+11] back-end execution engine based on the acyclic dataflow model. GraphX [GXD+14] offers a Scala-embedded GAS interface and runs on top of Spark. GraphX translates GAS workflows into traditional Spark procedural code that applies operations over RDDs. GraphLINQ [McS14] is a C# library for Naiad with LINQ-like operators optimised for graph processing. GraphLINQ offers an easy-to-use procedural programming interface in which workflows are expressed by manipulating relations storing graphs' vertices and edges.

To sum up, front-end frameworks provide high-level declarative abstractions that developers can use to express workflows. The frameworks translate workflows into low-level abstractions and execute them on back-end engines. In theory, front-ends should be able to translate workflows to several back-end execution engines, but in practice this is not the case: each front-end is coupled to a back-end (see Figure 2.4). I now show in a series of experiments that this coupling increases workflow makespan: no back-end execution engine outperforms all other.

2.2.3 No winner takes it all

Back-ends have built-in assumptions about the likely operating regime, and optimise their behaviour accordingly. This makes it difficult for developers to determine which back-end is best for a given workflow, input data set and cluster setup.

I illustrate this in a set of simple benchmarks that I execute on a heterogeneous local cluster and on a medium-sized cloud cluster [2]. The former represents a dedicated, fully controlled environment with low expected external variance, while the latter corresponds to a typical shared, multi-tenant data center. In all cases, I run frameworks over a shared Hadoop file system (HDFS) installation that stores input and output data. I either implement workflows directly against a particular back-end execution engine, or use a front-end framework with its corresponding native back-end.

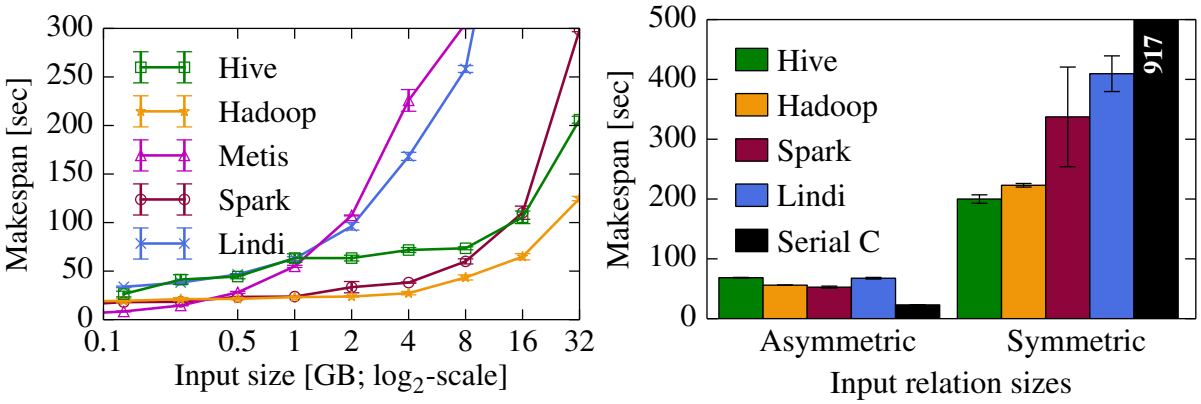
2.2.3.1 Performance

Batch data analytics workflows often consist of relational operators. In the following, I consider the behaviour of two operators in three isolated micro-benchmarks. I use the heterogeneous local cluster of seven nodes as an example of a small-scale data analytics deployment. In later experiments, I show that my results generalise to more complex workflows and to larger clusters. In all experiments I measure workflow *makespan* – i.e., the entire time to execute a workflow, including the computation itself and any data loading, pre-processing and output materialisation required.

Input size. A key question for a workflow developer is whether to leverage parallelism within a single-machine or across many machines. The answer to this question depends heavily on both input data size and back-ends' architectural design. A single-machine can be used efficiently for a workflow if the entire working set (including any overheads) fits into its main memory and the parallelism available in the machine is sufficient. It is important to understand the trade-offs between single-machine and distributed execution because workflows that can be processed on a single machine are common in practice: 40–80% of Cloudera customers' MapReduce jobs and 70% of jobs in a Facebook trace have $\leq$ 1GB of input [CAK12].

To investigate how input size affects system performance and when a single-machine architecture is best, I look at a simple string processing workflow in which I extract one column from a space-separated, two column ASCII input. This corresponds to a `PROJECT` query in SQL terms, but is also reminiscent of a common pattern in log analysis batch jobs: lines are read from storage, split into tokens, and a few are written back. I consider input sizes ranging from 128MB up to 32GB.

[2] See §4.1 for a description of the setup.



(a) `PROJECT` on a relation.

(b) `JOIN` two data sets.

Figure 2.7: Different systems perform best for simple queries. Lower is better; error bars show min/max of three runs.

In Figure 2.7a, I compare the makespan of this workflow on five different frameworks. Two of these are developer-friendly SQL-like front-ends (Hive, Lindi), while the others are back-end execution engines that require the developer to program against a low-level API (Hadoop, Metis and Spark). For small inputs ($\leq$ 0.5GB), the Metis single-machine MapReduce back-end performs best. Once the data size exceeds 0.5 GB, however, the distributed frameworks outperform it due to their ability to perform I/O in parallel. [3]

I/O efficiency. As input data size grows, Hive, Spark and Hadoop all surpass the single-machine Metis, not least since they stream data from and to HDFS in parallel. However, since the workflow consists of one operator and there is no data re-use. The Lindi front-end implementation for Naiad performs surprisingly poorly; I tracked this down to an implementation decision in the Naiad back-end, which uses only a single input reader thread per machine, rather than having multi-threaded parallel reads. Since the `PROJECT` benchmark is primarily limited by I/O bandwidth, this decision proves detrimental.

Data structure. Following, I consider a `JOIN` workflow. This workflow's output size is highly dependent on the structure of the input data: it may generate less, more, or an equal amount of output compared to its input. I therefore measure two different cases: *(i)* an input-skewed, *asymmetric* join of the 4.8M vertices of a social network graph (LiveJournal) and its 69M edges, and *(ii)* a *symmetric* join of two uniformly randomly generated 39M row data sets. In Figure 2.7b, I show the makespan of different front-ends and back-ends (plus a simple implementation in serial C code) for this workflow. The unchallenging asymmetric join (producing 1.28 million rows, 1.9 GB output size) works best when executed in single-threaded C code on a single machine, as the computation is too small to amortise the overheads of distributed

[3] Metis is not bottlenecked on computation, but on reading the data from HDFS. With the data already local, Metis performs best up to 2 GB input data.

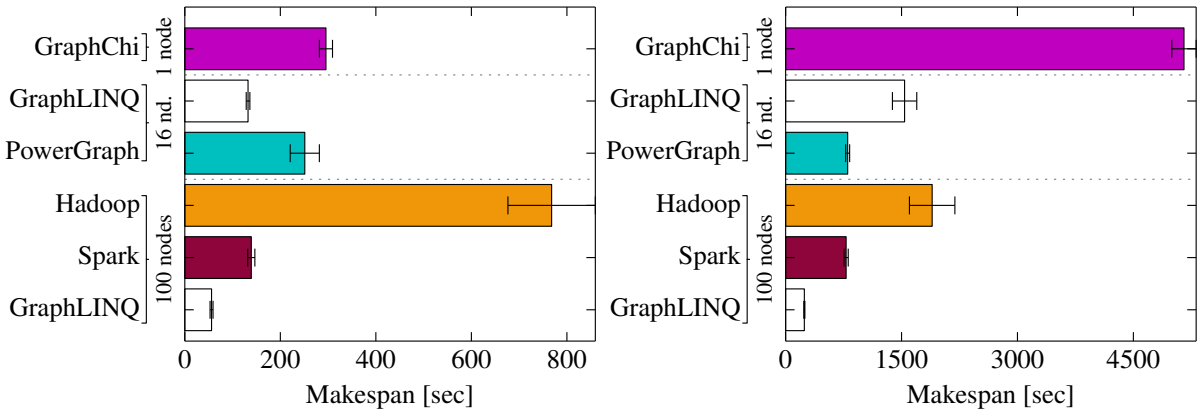
solutions. The far larger symmetric join (producing 1.5 billion rows, 29 GB output), however, works best when it is expressed in Hive and runs on Hadoop MapReduce, although I did need to manually optimise Hive to use a suitably high degree of concurrency. By default Hive uses the `CombinedHiveInputFormat` class to merge several input splits and hence reduce the total number of map tasks; however this also limits the potential parallelism which, in this case, is detrimental. I instead use the `HiveInputFormat` class and manually set the parallelism – a choice that data analysts would have difficulty to make, without knowledge of Hive's internals. Other systems suffer from inefficient I/O parallelism (e.g., Lindi uses a single-threaded writer), or have overhead due to constructing in-memory state and scheduling tasks sub-optimally (Spark). The takeaway here is that the best system may depend not only on the size of the data to be processed, but also on the nature of processing conducted, and its sensitivity to data skew.

Dataflow model. The choice of dataflow model can have a major impact on back-end performance and efficiency. For example, many common workflows involve iterative computations on graphs (e.g., social networks), which are ill-suited to execute on back-ends based on the bipartite dataflow model (§2.2.1.1). In the following, I compare different back-ends and front-ends running PageRank on social network graphs. I use a EC2 cluster (`m1.xlarge` instances) and I vary its size in order to determine systems' efficiency at different scales.

Many specialised graph processing systems are based on the bulk synchronous parallel or gather, apply, and scatter (GAS) models (§2.2.2). These systems cannot run many types of workflows, but can deliver significantly better performance on graph workflows than generalised back-ends. In Figure 2.8, I show the makespan of a five-iteration PageRank workflow on a small Orkut social network graph (3.0M vertices, 117M edges) and on a large Twitter graph (43M vertices, 1.4B edges). It is evident that graph-specialised systems have significant advantages for this computation: the GraphLINQ workflow implementation that runs on Naiad outperforms all other systems.[4] PowerGraph also performs very well, since its vertex-centric sharding reduces the communication overhead that dominates PageRank makespan.

Engineering choices. Engineering choices such as targeted scale and back-end implementation language also have an impact. For example, Hadoop is designed to run on clusters of thousands of nodes [HAD16], Spark on clusters of hundreds of nodes [ZCD+12], and PowerGraph targets dozens of powerful machines. Different techniques make sense at different scales. Hadoop, for example, piggy-backs task launch messages onto coarsely spaced heartbeats to avoid incast at the master node, leading to widely documented task spawn overhead [OPR+13], but improving scalability. In contrast, Spark eagerly launches tasks as soon as they are scheduled. Powergraph provides no fault tolerance because it is designed to run on tens of machines, setups in which workflows are less likely to be affected by failures.

[4] Only the GraphLINQ front-end for Naiad is shown here; Lindi is not optimised for graph computations and performs poorly.



(a) Orkut (3.0M vertices, 117M edges).

(b) Twitter (43M vertices, 1.4B edges).

Figure 2.8: Makespan of PageRank on social network graphs in different back-ends and front-ends. Makespan varies depending on scale; lower is better; error bars: $\pm\sigma$ of 10 runs.

Other seemingly unimportant engineering decisions such as which programming language to use can also significantly affect performance. For example, many back-ends are implemented in managed languages with garbage collection. These back-ends are highly sensitive to garbage collector configuration parameters. In my experiments, I frequently observed stalls when large heaps had to be garbage-collected [5]. By contrast, other systems have been implemented in unmanaged languages. For example, PowerGraph is written in C++ and utilises machines' compute resources efficiently.

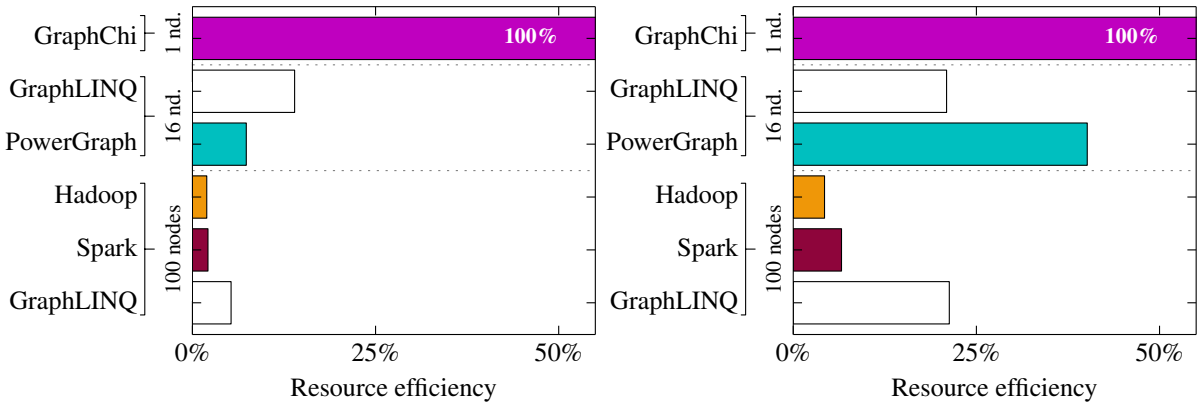
2.2.3.2 Resource efficiency

The fastest back-end is not always the most resource efficient. While PageRank implemented in GraphLINQ and running on 100 Naiad nodes has the lowest runtime in Figure 2.8b, PowerGraph performs better than GraphLINQ when using only 16 nodes (due to its improved sharding).[6] Moreover, when the input graph is small, GraphChi performs only 50% worse than Spark on 100 nodes, and only slightly worse than PowerGraph on 16 nodes, despite using only one machine (see Figure 2.8a). In such situations, it may be worthwhile to wait for longer for workflows to complete, and save resources.

Resource efficiency is a measure of how much additional resources are used during execution. I compute this metric by taking the aggregate number of machine seconds consumed by the workflow, and normalise it to the best single-machine execution. For example, a workflow running for exactly $30s$ on a back-end deployed on 100 machines incurs an aggregate runtime of $3,000s$. If the best single-machine back-end completes the same workflow in $1,500s$, the resource efficiency is 50% when running on 100 machines.

[5] In some cases, garbage collection even manifested itself as a failure: garbage-collecting a 64 GB JVM heap took more than 60s and triggered system-level timeouts.

[6] PowerGraph requires the number of nodes to be a power of two, but running it on 32 or 64 nodes showed no benefit over running it on 16 nodes.



(a) Orkut (3.0M vertices, 117M edges). **(b)** Twitter (43M vertices, 1.4B edges).

Figure 2.9: Resource efficiency of different data processing systems running PageRank on a EC2 cluster.

In Figure 2.9, I show the resource efficiency for the PageRank workflow by normalising systems' resource usage to that of GraphChi. Only PowerGraph comes close to GraphChi in terms of total machine time used, but still consumes significantly more resources. PowerGraph's resource use is between 2.5× (Twitter) and 8× (Orkut) that of GraphChi.

2.2.4 Workflow managers

In the previous subsection, I argued that the "best" system for a given workflow varies considerably. The right choice – i.e., the fastest or most efficient system – depends on the workflow, the input data size, structure of the data, engineering decisions, dataflow models and parallelism available.

Workflow managers are systems that tackle this problem. They receive workflows from developers, generate jobs, and manage jobs through their lifetime (e.g., restart jobs if failures occur). FlumeJava [CRP+10] is one such workflow manager. FlumeJava defers execution of operations on Java parallel collections until runtime. These operations' implementation is abstracted away from developers and can range from a local iterator to a MapReduce job, depending on input data size. FlumeJava considers multiple execution strategies, but only targets the MapReduce back-end execution engine and does not leverage the other back-ends that have different properties. Similarly, Pig only supports the Hadoop MapReduce back-end execution engine [ORS+08], whereas Oozie does not optimise workflows at runtime [OOZ16]. QoX [SWC+12] combines databases and data processing engines by separating logical operations and physical implementation, but, is limited to only handle extract, transform and load workflows (ETL).

2.2.5 Data processing systems summary

The increasingly diverse workflows and the different performance characteristics of data processing systems make it difficult for developers to decide which system to use. In the previous subsection, I demonstrated that no single front-end or back-end systematically outperforms all others. I also highlighted that systems peformance depende os:

- **input data size**, as single machine back-ends outperform distributed back-ends for small inputs;
- **structure of the data**, since skew and selectivity impact I/O performance and work distribution;
- **underlying dataflow models** used by back-ends, as some may not be well-suited for iterative computations (e.g., bipartite dataflows), and specialised systems often operate more efficiently;
- **engineering decisions**, with e.g., overheads due to implementing back-ends in managed languages, and data loading input costs varying significantly across frameworks.

Understanding when and how these properties and decisions affect workflow performance requires detailed, in-depth knowledge of systems' inner workings. It is unrealistic to expect the majority of workflow developers to have this knowledge. Moreover, the information may not be available at workflow implementation time, which motivates dynamically deciding at runtime which combination of back-ends to run the workflow on. In this dissertation, I show how workflow makespan can be reduced and resource efficiency improved by executing computations with a workflow manager that generates back-end workflow code at runtime and dynamically uses several back-ends for a given workflow. In Chapter 3, I describe Musketeer, my proof-of-concept workflow manager that achieves this with its approach that *decouples* front-end frameworks from the back-end execution engines. In Chapter 4, I evaluate how fast workflows complete in Musketeer compared to running them in a single back-end.

2.3 Cluster scheduling

Cluster management systems such as Mesos [HKZ+11], YARN [VMD+13], Borg [VPK+15], and Kubernetes [KUB16] automatically share and manage physical cluster resources. However, the isolation techniques used by current virtualization solutions (e.g., memory limits, disk quotas) do not offer performance isolation. In practice, when two tasks are co-located on a machine they will likely affect each other and cause application-level performance variation. This happens because the underlying hardware – the machines, the network and the storage – is still fundamentally shared among tasks.

The role of the *cluster scheduler* is to place tasks on cluster machines and to manage tasks' resource allocations. The scheduler must place tasks in such a way that machines are shared efficiently and application-level performance is not significantly affected. These demands are increasingly more difficult to meet as workloads become increasingly diverse and clusters grow in size.

In the following sections, I discuss the requirements typical workloads impose on cluster schedulers (§2.3.1). Following, I describe the main cluster scheduler architectures and present their properties (§2.3.2). Finally, I present a summary of state-of-the-art cluster scheduling (§2.3.3).

2.3.1 Scheduler requirements

I outline below the challenges cluster schedulers must overcome in order to efficiently utilise clusters and meet the demands of the workloads they execute.

2.3.1.1 Hardware heterogeneity

Data center clusters are built using commodity machines purchased in bulk. Nevertheless, cluster hardware is more heterogeneous than one might expect: machines are replaced upon failure, hardware upgrades are rolled out regularly, and resources are intentionally diversified [TMV+11; BCH13; MT13].

Most clusters have memory and storage of different latency and bandwidth, and run at least three processor generations. Hence, it is common to have up to 30 different machine configurations [DK13]. An analysis of a publicly available Google trace found that a typical cluster has three different machine platforms and ten machine specifications [RTG+12; Sch16]. Similarly, a study of Amazon's EC2 infrastructure found that `m1.large` instances use five different CPU models. This diversity can lead to a 60% workload runtime variation even when the same type of EC2 instances are used [OZN+12], or up to 100% runtime increase when different CPU generations are used [Sch16, §2.2.2].

Workload performance is affected even more in clusters that deploy specialised hardware such as GPU accelerators, InfiniBand and flash storage. In order to efficiently utilise this hardware, the cluster scheduler must place on it low-priority tasks when high-priority tasks do not fully utilise it. The scheduler must also make sure that low-priority tasks do not affect high-priority tasks' performance.

Finally, the cluster scheduler must also be able to leverage hardware heterogeneity to reduce power usage. Utilising more power-effective hardware when workloads service level objectives allow it, can lead to improvements of up to 20% in power efficiency [NIG07; Sch16].

2.3.1.2 Task co-location interference

Tasks differ significantly in their resource requirements (see Figure 2.2). A scheduler that simply bin-packs tasks to use all the CPUs, memory and disk I/O will achieve high resource utilisation at the cost of poor and variable application-level performance (e.g., high latency and low number of served queries per second). When two or more tasks execute concurrently on a machine, they can interfere on the fundamentally shared intra-machine resources (e.g., disk bandwidth, memory bandwidth, cache, shared operating system data structures) and inter-machine cluster resources (e.g., network links).

Schwarzkopf shows in a study of task co-location that highly-optimised SPEC CPU2006 compute kernels suffer degradations of up to 2.3× when two tasks are co-located on a machine. These results also apply to data center applications (e.g., PageRank, strongly connected components, image classification) whose performance degrade up to 2.1× compared to when they run on otherwise idle machines [Sch16, §2.2.3]. Similarly, Mars *et al.* found a 35% degradation in application-level performance for co-located Google workloads [MTH+11]. Such co-location interference also affects latency critical production applications and can cause service level objectives (SLOs) violations, and latency degradations of up to 3× [LCG+15].

Each task competes for the shared resources, but certain tasks make better neighbours, while others can cause significant performance degradation. The main challenge for the cluster scheduler is to place and manage tasks in such a way that they do not significantly interfere. Two different solutions have been proposed for this problem: *(i) non-controlling* solutions that consider how a task would be affected by other running tasks before palcing it on a machine, and *(ii) controlling* solutions that use mechanisms to pace resources utilisation of low-priority tasks in order not to affect high-priority latency critical tasks.

Non-controlling interference avoidance. Paragon [DK13] is a scheduler that profiles tasks to discover how they perform on different machine architectures and how much interference on various resources affects them. Paragon inputs these profiling results into a collaborative filtering algorithm that classifies submitted tasks and identifies similarities with tasks previously scheduled. Finally, Paragon greedily places submitted tasks on machines with well-suited hardware on which tasks do not interfere. In practice, task resource utilisation can vary greatly as tasks go through different stages or as the load varies because of diurnal patterns [CAB+12]. However, Paragon's profiling step can miss such resource utilisation variations because it only lasts for several seconds. The Quasar [DK14] scheduler uses similar techniques, but shifts from a reservation-centric approach in which developers ask for a fixed amount of resources for each task, to a performance-centric approach in which developers express performance requirements (e.g., queries per second, query latency). Quasar conducts more extensive profiling than Paragon: it also studies application scale-up and scale-out behaviour. Quasar uses these profiling results to determine the least amount of resources required to meet applications' performance requirements, given currently available machines and active workloads.

Controlling interference avoidance. Non-controlling interference avoidance techniques either throttle the number of tasks that execute on a machine, migrate tasks, or scale-out applications in order to maintain performance. However, these techniques may not be suitable for high-priority latency critical tasks. Task throttling stops new tasks from being placed on machines, but does not decrease interference among running tasks. Task migration can cause application downtime or may require state to be recomputed, which can increase application latency.

Heracles [LCG+15] is a feedback-based controller that actively reduces task co-location interference. It dynamically manages several software and hardware isolation mechanisms to reduce interference on cores, network and last-level-cache (LLC). Heracles uses Linux cpuset cgroups to pin tasks to cores, the Linux qdisc scheduler with hierarchical token bucket queuing to enforce bandwidth limits on outgoing task traffic, and Intel's Cache Allocation Technology (CAT) hardware to partition the shared LLC. Heracles combines these techniques to achieve 90% average machine utilisation without violating latency SLAs for high-priority tasks.

To sum up, cluster scheduler must co-locate tasks to achieve high cluster utilisation, but they must do so without affecting application-level performance. An ideal scheduler would take into account co-location interference when it places tasks, but it would also use controlling techniques to reduce interference while tasks execute.

2.3.1.3 Multi-dimensional resource fitting

Many cluster schedulers assume that tasks have uniform resource requirements and thus statically partition machines into a fixed number of "slots", in which they execute tasks [DG08; IPC+09; OWZ+13; VMD+13; DDK+15; KRC+15]. However, nowadays' clusters execute different types of tasks (e.g., interactive, batch and service) which have significantly different resource needs. In Figure 2.10, I show the normalised mean CPU and memory usage for 10,000 tasks picked at random from a publicly available Google trace of one of their clusters [RTG+12]. Task CPU and memory usage vary greatly and there is no correlation between the two.

Cluster schedulers that statically partition machines into slots are not well suited to handle such diverse workloads because they implicitly assume that tasks have similar resource requirements (i.e., place tasks into equally-sized slots). They fundamentally cannot achieve high resource utilisation and good application performance.

To sum up, the cluster scheduler must dynamically allocate resources to tasks, and it must achieve a **good bin-packing** on different resource dimensions in order to neither under-utilise nor over-utilise resources.

2.3.1.4 Data locality

Data locality is essential for many tasks that run in today's clusters. Data-intensive tasks can become "stragglers" if their data locality preferences are ignored. Tasks that read data remotely

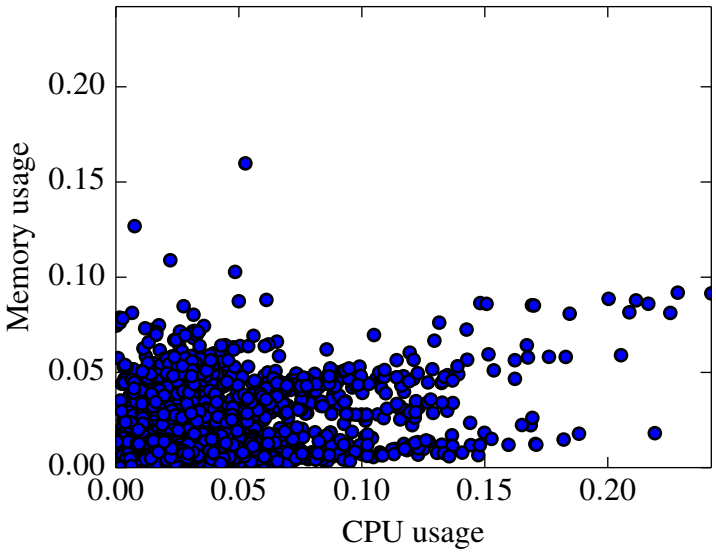


Figure 2.10: Analysis of task CPU versus memory usage in the Google trace. The values are normalised to the highest value present in the trace for the given type of metric.

create load on the cluster network and can affect each other. Schedulers that do not take into accound data locality could co-locate tasks that do not read local data next to tasks that read non-replicated local input data [ZKJ+08; AGS+13]. Such placements could increase makespan because tasks would compete for disk bandwidth: non-local data tasks would write output, and local data tasks would read input and write output.

Cluster schedulers aim to place tasks close to input data to alleviate disk and network bandwidth interference, but other complementary techniques also exist: *(i)* some schedulers delay task placement until better placement options become available [ZBS+10; HKZ+11], *(ii)* other schedulers consider pre-empting already running tasks in order to improve data locality [IPC+09], and *(iii)* distributed files systems automatically increase the replication of popular data in the hope of increasing the likelihood of achieving data locality [AAK+11].

Up until recently, disks provided higher bandwidth than data center host network links. However, today's data centers use 10 Gbps, full-bisection bandwidth Ethernet which offers an order of magnitude higher bandwidth than disks. This increase in bandwidth combined with new low-latency switches have made the latency and bandwidth differences between local and remote disk accesses insignificant [AGS+11]. However, high task data locality still remains a key requirement for cluster schedulers because data centers nowadays deploy systems that store data in memory [ORS+11; ZCD+12; LGZ+14]. Ignoring data locality in such systems leads to significant network traffic, which in turn creates network congestion that affects application performance [GSG+15].

To sum up, in order to reduce network and disk interference, cluster schedulers must place tasks such that they read a high fraction of their input locally. Applications placed by such schedulers are less affected by network congestion and have better and more predictable performance.

2.3.1.5 Placement constraints

Data center clusters are built with increasingly more heterogenous hardware (§2.3.1.1) and run more and more diverse tasks (§2.1). Some tasks can only run on machines that have certain properties (e.g., service tasks that run web servers require machines with public IP addresses), or may benefit if they run on specialised hardware (e.g., machine learning tasks complete faster if they are placed on GPUs). Cluster schedulers represent such task requirements as *placement contraints* that restrict the set of machines on which tasks can be placed. There are three types of placement constraints:

- **Hard constraints** specify requirements that machines *must* satisfy. Tasks with hard constraints must not be placed as long as there are no machines with available resources that satisfy the constraints. Hard constraints are very common; over 50% of Google's tasks have simple hard constraints. These typically specify constraints on machine attributes such as kernel version and clock speed[SCH+11; RTG+12].

  Cluster schedulers adopt different techniques when they place tasks with hard constraints. Some cluster schedulers (e.g., Sparrow, Borg) sample machines until they find several that satisfy the constraints [OWZ+13; VPK+15]. Others (e.g., YARN), communicate with application controlers (i.e., job managers in YARN) to inform them if task hard constraints can be fulfilled [VMD+13].

- **Soft constraints** do not have to necessarily be satified for the cluster schedulers to place tasks. Tasks can execute, possibly with degraded performance, even when their soft constraints are not satisfied. For example, TensorFlow [ABC+16] machine learning tasks are typically executed on GPUs, but they can also run on CPUs, albeit they take longer to complete.

  Cluster schedulers use different techniques to support soft constraints. Quincy models soft constraints as task placement preferences [IPC+09]. It creates a flow network that contains a node for each cluster task and machine. Quincy directly connects with preference arcs task nodes to the nodes of the machines that satisfy their soft constraints. Following, it runs a min-cost flow optimisation over the network that discovers task placements. Quincy explores the trade-offs satisfying task data locality soft constraints, costs of leaving tasks unscheduled, or task preemption costs (see §5.1.2).

  Quasar does not allow users to express low-level soft constraints for hardware, but instead requires them to express high-level application soft constraints (e.g., latency, throughput, queries per second). It tries to satisfy these constraints by automatically adjusting how many resources applications receive.

- **Complex constraints** combine hard and soft constraints to model complex requirements that two or more tasks or machines have. *Task affinity* and *task anti-affinity* are two popular types of complex constraints. Task affinity constraints model requirements for

placing two or more dependent tasks on a resource (e.g., a task that runs a web server and another task that runs a database used by the web server must be placed on the same machine). By contrast, task anti-affinity constraints model requirements for placing tasks on distinct resources. For example, some jobs may require tasks to be placed on different machines or racks in order to decrease downtime likelihood in case of hardware failures.

Only few cluster schedulers support complex constraints. Borg only allows users to specify simple task anti-affinity constraints [VPK+15]. Nevertheless, 11% of tasks from a publicly available trace of a Google cluster have anti-affinity constraints [SCH+11; RTG+12]. Alsched [TCG+12] and tetrisched [TZP+16] are the only schedulers that fully support complex constraints. However, complex constraints are expensive to support, and thus these schedulers trade-off task placement latency and scalability for complex constraints support.

To sum up, support for the different types of constraints is appealing because among others, constraints guide schedulers to place tasks such that tasks achieve improved performance and fault tolerance. However, task constraints can significantly increase task placement latency. For example, Google's scheduler task placement latency increases by up to 6× when constraints are enabled [SCH+11]. An ideal scheduler would support constraints without significantly increasing task placement latency and without scalability degradation.

2.3.1.6 Resource estimation and reclamation

Many cluster managers statically partition machines into "slots" in which they execute tasks. These managers use slots as a straw man's admission control mechanism to avoid oversubscribing machines (i.e., no more tasks can execute than how many slots machines are partitioned into) [DG08; IPC+09; OWZ+13; DDK+15; KRC+15]. However, in practice not all tasks use the same amount of resources. In Figure 2.10, I show that tasks have widely different resource requirements. Cluster schedulers that execute such diverse workloads in "slots" either oversubscribe machines because they co-locate too many resource-intensive tasks, or underutilise machines because they co-locate tasks with low-resource requirements.

In order to address these limitations several cluster schedulers expect users to specify resource requirements at task submission time [HKZ+11; VMD+13; VPK+15; KUB16]. These schedulers use the requirements to allocate resources and to bin-pack tasks on machines in such a way that CPU and memory are almost fully utilised. However, in practice, users find it difficult to predict how many resources tasks require. To make matters worse, cluster managers kill tasks that utilise more resources than requested. Users are incentivised to overestimate task resource needs. In a production cluster at Twitter managed with Mesos, 70% of the jobs overestimate requirements by up to 10×, while 20% underestimate requirements by up to 5× [DK14]. Similarly, users significantly overestimate task resource requirements for tasks they run on Google's

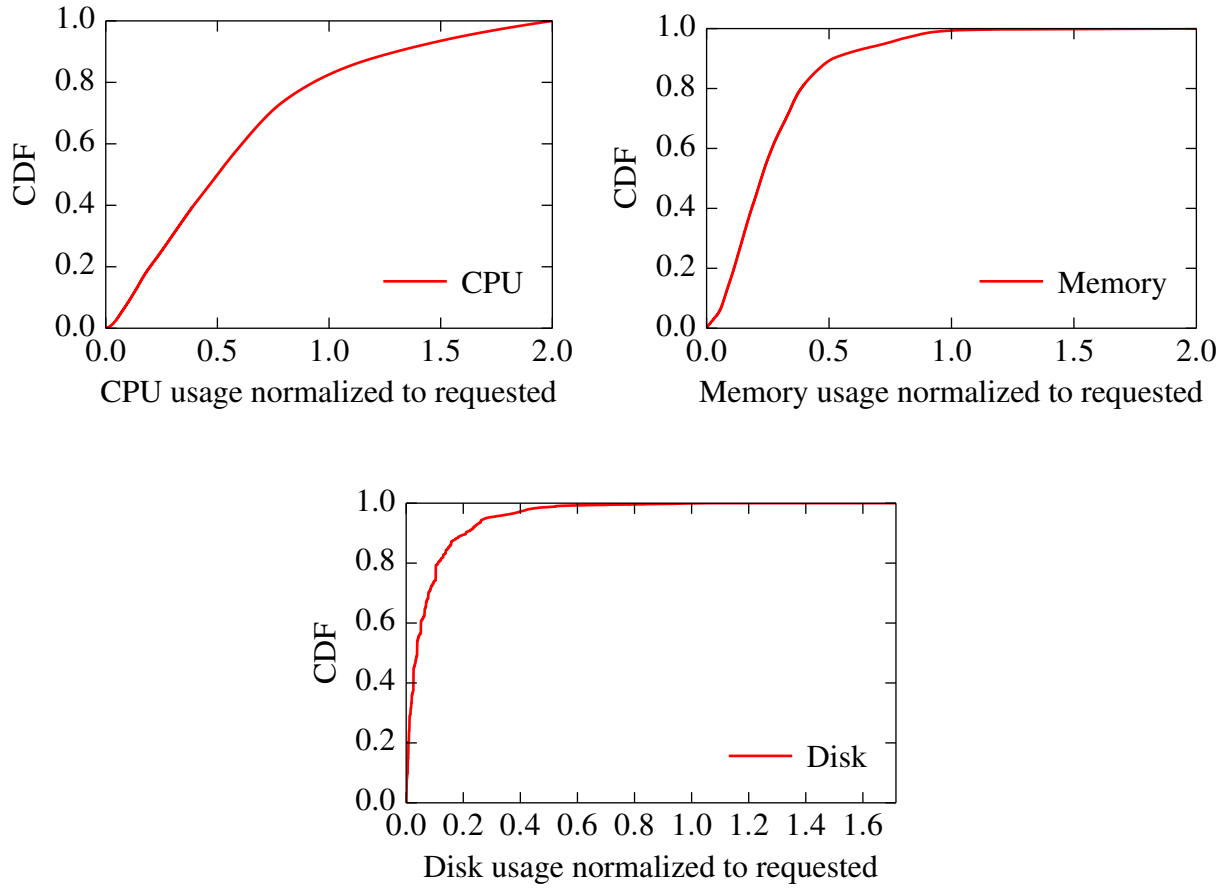


Figure 2.11: Resource requests normalised to usage from a Google cluster.

Borg cluster manager. In Figure 2.11, I show task CPU, memory and local disk usage normalised to resource request from a Google cluster [RTG+12]. In this cluster, 50% of the tasks use less than 50% of requested CPU, 90% of the tasks use less than 50% of requested memory, and 90% of the tasks use less than 20% of requested local disk space. The Borg cluster manager *reclaims resources* and dynamically adjusts resource reservations if tasks underutilise allocated resources [VPK+15]. Borg achieves aggregate CPU utilisation of 25-35% and aggregate memory utilisation of 40% even though users significantly overestimate task resource requirements.

Other cluster managers automatically *estimate resource requirements*. Quasar [DK14] profiles tasks for several seconds to estimate the impact scaling up and scaling out tasks has on application performance. However, task resource utilisation varies greatly during task lifetime [CAB+12]. Tasks have periods when they communicate and compute, but also have periods when they wait for other tasks to do work. Quasar may miss such resource utilisation variations in its short task profilling step. By contrast, Jockey [FBK+12] requires users to specify deadlines by when jobs must complete. Upon job execution, Jockey dynamically and automatically adjusts task resource allocations such that as many jobs as possible meet deadlines. While, this technique can work for tasks that must complete, it cannot be applied to long-running service tasks.

Schedulers that support either automatic resource estimation or dynamic resource reclamation underutilise clusters. An ideal scheduler should use both techniques to better predict users' resource needs, and to dynamically adjust resource reservations at runtime.

2.3.1.7 Fairness and guaranteed access to resources

Virtualization enables data center operators to increase their revenues by providing on demand hardware resources to third party users. Underlying data center resources (e.g., machines, network and storage) are shared by users who want to fully utilise the resources they pay for. However, some users may not get what they pay for because others aggressively use resources (e.g., saturate network links, use all disk bandwidth).

Data center operators strive to offer access to resources to all paying users. Thus, cluster schedulers must ensure that resources are fairly utilised at both network-level and machine-level.

Network-level fairness. Two approaches can be used to provide network-level guarantees and fairness: *(i)* end host pacing, and *(ii)* sophisticated placement of tasks and virtual machines. For example, Oktopus [BCK+11] uses end host-based rate enforcement to achieve max-min fair share for network traffic between virtual machines. Similarly, FairCloud [PKC+12] offers per-source fair sharing for traffic to the data center network root and per-destination fair sharing for traffic from the root. Finally, the Silo [JSB+15] network architecture offers guaranteed network bandwidth, predictable packet delay and guaranteed burst allowance. Silo uses end host packet packing and a VM placement algorithm to explore the trade-offs between network bandwidth and queuing delay.

Machine-level fairness. Cluster managers are also responsible for fairly sharing machine resources among users. Some state-of-the-art cluster schedulers apply admission control to achieve this. For example, the Hadoop Fair Scheduler ensures that users cannot run more map and reduce tasks than an allocated share tasks [HFS16]. Apollo [BEL+14] uses a token-based mechanism to allocate job capacity, and offers probabilistic slot fairness.However, neither scheduler guarantees fair shares in the presence of long-running tasks: when clusters are over-subscribed, users may have to wait until long-running tasks complete to get a fair share of resources. By contrast, the Quincy scheduler enforces fair shares in the presence of long-running tasks because it preempts tasks that cause users to exceed their share of resources [IPC+09]. However, temporary unfairness is still possible because Quincy does not preempt tasks that already ran for a while in order to guarantee task progress.

The schedulers I discussed so far try to fair share cluster slots, but schedulers that run tasks in fixed sized slots do not achieve high cluster utilisation (§2.3.1.6). By contrast, Dominant Resource Fairness (DRF) [GZH+11] considers the problem of allocating fair resource shares to users that run tasks with requirements for different types of resources (e.g., CPU, memory,

disk). A task request request is "dominant" if it amounts to the largest resource share the task requests across all resource types. DRF prioritises tasks with low dominant resource requests, and thus offers max-min fairness for dominant resources.

To sum up, state-of-the-art cluster schedulers offer resource allocation guarantees or fairness for either the network or machine resources, but not both. However, an ideal cluster scheduler should fairly share both, even in the presence of multi-dimensional resource requirements and task placement constraints.

2.3.1.8 Scalability and low scheduling latency

High-quality task placements lead to higher machine utilisation [VPK+15], more predictable application performance [ZTH+13; DK14], and increased fault tolerance [SKA+13]. However, high-quality placements require schedulers that are aware of hardware heterogeneity (§2.3.1.1), task co-location interference (§2.3.1.2), achieve good bin-packing on different resources (§2.3.1.3), provide task data locality (§2.3.1.4), support placement constraints (§2.3.1.4) and accurately estimate the amount of resources tasks require (§2.3.1.6). Schedulers have to run slow, algorithmically complex optimisations in multiple dimensions to support all these features.

Nevertheless, placement latencies of multiple seconds or even minutes are unacceptable for many workloads. For critical service tasks, a failure recovery delay of minutes due to scheduling can seriously impact availability [SKA+13]. Moreover, many short, interactive batch tasks only run for a short time: for example, 50% of SCOPE tasks at Microsoft take less than 10s [BEL+14, §2.3] and 30% of tasks from one of Google's clusters take less than 200s [RTG+12].

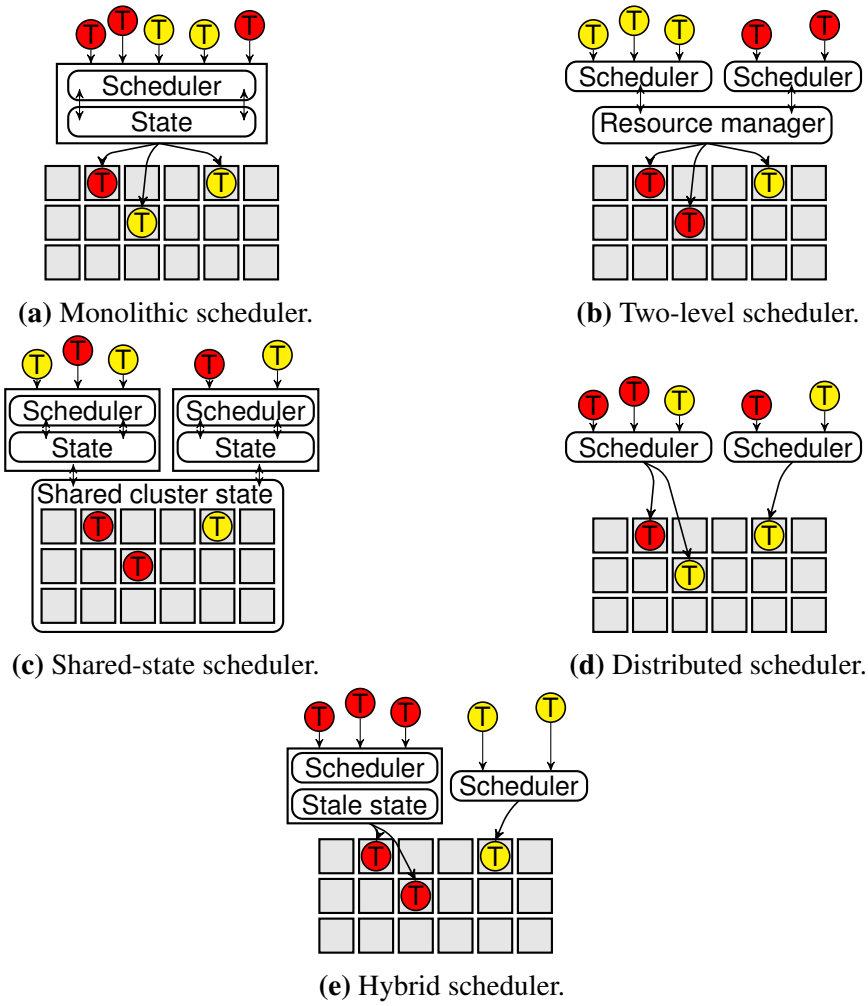
Thus, it is critical the scheduler places tasks with low placement latency for maintaining high cluster utilisation, keeping task makespans low, and meeting user expectations. Sophisticated schedulers choose high-quality placements, but the algorithms they use are computationally expensive and slow. For example, Quincy [IPC+09], which runs a minimum-cost flow optimisation over a flow network, is widely acknowledged to choose high-quality placements, but its placement latency increases to minutes at scale [GSG+16, §2.2; OWZ+13, §9; GZS+13, §8; GSW15, §5; BEL+14]. Quincy cannot choose any placements for incoming tasks while the optimisation runs. This leaves cluster resources idle and increases task placement latency even when resources are available. An ideal scheduler should be able to provide low latency placement latency without sacrificing task placement quality.

2.3.2 Cluster scheduler architecture

All today's prevalent cluster scheduler architectures trade-off between placement quality and placement latency. In Figure 2.12, I show a high-level view of these architectures, and in Table 2.2 I show what arhitectures state-of-the-art schedulers have and summarise what features they support.

Scheduler	Workload	Architecture	Hardware heterogeneity	Co-location interference	Multi-resource fitting	Data locality	Placement constraints	Resource estimation	Fairness	Low latency at scale
LATE [ZKJ+08]	MapReduce	Centralised	✓	–	–	–	–	–	–	–
HFS [ZBS+10]	MapReduce	Centralised	–	–	✓	–	–	✓	–	–
H-DRF [BCF+13]	MapReduce	Centralised	–	✓	–	–	–	✓	–	–
Quincy [IPC+09]	Dryad tasks	Centralised	–	–	✓	✓	–	✓	–	–
Jockey [FBK+12]	SCOPE	Centralised	–	–	–	–	✓	–	–	–
Apollo [BEL+14]	SCOPE	Distributed	–	✓	✓	–	–	(✓)	✓	✓
alsched [TCG+12]	Simulation	Centralised	✓	–	✓	✓	–	–	–	–
Sparrow [OWZ+13]	Spark tasks	Distributed	–	–	–	–	–	(✓)	✓	✓
KMN [VPA+14]	Spark tasks	Centralised	–	–	✓	–	–	–	–	–
Hawk [DDK+15]	Spark tasks	Hybrid	–	–	–	–	–	–	✓	✓
Eagle [DDD+16]	Spark tasks	Hybrid	–	–	–	–	–	–	✓	✓
Mercury [KRC+15]	Data-processing tasks	Hybrid	–	–	–	✓	–	(✓)	✓	✓
YAQ [RKK+16]	Data-processing tasks	Hybrid	–	–	✓	–	–	✓	✓	✓
Choosy [GZS+13]	Mixed	Centralised	–	(✓)	✓	✓	–	✓	–	–
Paragon [DK13]	Mixed	Centralised	✓	✓	–	–	–	–	–	–
Whare-Map [MT13]	Mixed	Centralised	✓	✓	–	–	–	–	–	–
Quasar [DK14]	Mixed	Centralised	✓	✓	✓	(✓)	–	✓	–	–
Bistro [GSW15]	Mixed	Centralised	(✓)	–	✓	✓	✓	–	(✓)	(✓)
tetrisched [TZP+16]	Mixed	Centralised	✓	–	✓	✓	✓	–	(✓)	–
Mesos [HKZ+11]	Mixed	Two-level	–	–	✓	✓	–	–	✓	–
Yarn [VMD+13]	Mixed	Two-level	–	–	✓	✓	–	–	✓	–
Omega [SKA+13]	Mixed	Shared-state	✓	–	✓	✓	(✓)	–	(✓)	✓
Tarcil [DSK15]	Mixed	Shared-state	✓	✓	✓	✓	–	✓	–	✓

Table 2.2: Existing cluster schedulers and their properties. Ticks in parentheses indicate that the scheduler could attain this property via extensions.



(a) Monolithic scheduler.

(b) Two-level scheduler.

(c) Shared-state scheduler.

(d) Distributed scheduler.

(e) Hybrid scheduler.

Figure 2.12: Comparison of different cluster scheduler architectures. Red circles represent long-running tasks, yellow circles correspond to short-running tasks, grey boxes are machines, and schedulers that store private cluster state are shown in rectangular boxes.

Monolithic schedulers. Complex scheduling algorithms that choose high-quality placements often need complete, up-to-date information about cluster state (e.g., machine utilisation, running tasks, unscheduled tasks). Monolithic schedulers are a great fit for these complex algorithms because: *(i)* they manage and place tasks for entire clusters, *(ii)* use same scheduling logic to choose placements for all different types of tasks, and *(iii)* store information about the cluster state in a *centralised* component (see Figure 2.12a).

However, the simplicity of monolithic, centralised scheduler architecture comes at a cost: increased task placement latency [IPC+09; OWZ+13]. In practice, monolithic schedulers often stop early their complex scheduling algorithms or use simple heuristics to place tasks. For example, Google's Borg centralised scheduler uses a multi-dimensional resource model to determine feasibility of task placement on machines, and a greedy cost-based scoring algorithm to choose among feasible machines. Borg does not asses feasibility for each machine, but scores randomly sampled machines until a termination condition is met [VPK+15, §3.4]. Thus, Borg sacrifices placement quality for placement latencies of seconds. Likewise, Facebook's

Bistro [GSW15] runs expensive machine scoring computations (collaborative filtering and path selection), but uses simple greedy algorithms for task placement in order to reduce latency.

Alsched [TCG+12] and TetriSched [TZP+16] are centralised, monolithic schedulers that model task placement as a Mixed Integer Linear Programming (MILP) problem. Both schedulers support complex placement constraints, facilitate planning ahead in time of task resource utilisation, and choose high-quality placements. However, at scale their placement latency increases to seconds, and thus they both stop early their scheduling algorithms, and trade-off placement quality for latency.

Finally, Quincy [IPC+09] is a centralised, monolithic scheduler that reconsiders the entire workload each time it places tasks. Quincy achieves optimal task placement for its scheduling policy. However, Quincy is widely considered unscalable: "Quincy [...] takes over a second [...], making it too slow" [OWZ+13, §9], "Quincy sacrifices scheduling delays for the optimal schedule" [GSW15, §5], "Quincy suffers from scalability challenges when serving large-scale clusters" [BEL+14, §6], "[Quincy's] decision overhead can be prohibitively high for large clusters" [DSK15, §1].

Two-level schedulers. Data center clusters run an increasing suite of data processing systems that execute batch, stream, graph and iterative workflows (§2.1). Hindman *et al.* [HKZ+11] notice these different types of workflows, and the programming models and communication patterns they create give rise to diverse scheduling needs. They argue that a monolithic, centralised scheduler is unlikely to provide an API that is flexible enough to express the different scheduling policies required by all these systems, and if it were possible to build such a scheduler its complexity would negatively affect scalability. Instead, Hindman *et al.* introduce the *two-level architecture* [HKZ+11] (see Figure 2.12b). Cluster managers that implement the two-level architecture manage resources, but delegate scheduling to other systems. Mesos is the first two-level cluster scheduler [HKZ+11]. Applications that run in Mesos-managed clusters receive resource reservation offers from Mesos. The applications must decide how many resources they need, which resource reservation offers to accept, and which tasks to allocate to these reservations. Similarly, the YARN [VMD+13] resource manager has a two-level architecture, but in contrast to Mesos, applications make resource reservation requests to the resource manager.

Both Mesos and YARN have three key drawbacks. First, they *hide cluster state information* because system schedulers only have information about the resources they reserve or they are offered. The schedulers do not have access to other relevant information such as: what other tasks run on the machines on which they are offered resources, which other machines have available and possibly better suited resources. Second, tasks can experience *priority preemption*: high-priority tasks do not preempt low-priority tasks that are placed by another scheduler because the scheduler that places high-priority task is not aware of the existence of these low-priority tasks. Finally, systems that must start multiple tasks simultaneously could *hoard resources*. For example, MPI jobs and stateful graph processing systems benefit if all job tasks start simulta-

neously. Their schedulers are incentivised to accumulate resource reservations until they can execute all tasks, but these resources are wasted because no tasks execute until the schedulers accumulate enough resources.

Shared-state schedulers. Like the two-level architecture, the *shared-state scheduler architecture* was developed to effectively schedule different types of workflows in shared clusters, and to enable system developers to easily build custom schedulers.

Google's Omega is the first cluster manager to use a shared-state scheduler architecture [SKA+13]. In Omega multiple schedulers run in parallel, and each schedules a subset of the workload (see Figure 2.12c). Schedulers implement different policies that use weakly consistent replicas of the entire cluster state to choose placements. In contrast to two-level schedulers, Omega has access to the entire cluster state and can directly modify it with optimistically-concurrent transactions.

Two or more schedulers can *directly* or *indirectly* choose conflicting placements in Omega. Schedulers choose directly conflicting placements if they simultaneously decide to place one or more tasks on the same resource. In such cases, only one scheduler is allowed to place a task, while others have to retry. Shared-state schedulers can choose indirectly conflicting placements if they have incompatible features. For example, a co-location interference aware scheduler could place a web service task on a machine on which the task does not interfere with other running tasks. However, a low-latency co-location interference unaware scheduler could later place interfering tasks on the same machine, which would increase serving latency and decrease web serving rate.

Apollo is a shared-state cluster scheduler that assumes that conflicts are not always harmful [BEL+14]. In contrast to Omega, Apollo does not eagerly resolve conflicts, instead it first dispatches tasks to machines, and then does conflict resolution. Apollo's schedulers can simultaneously place tasks on a machine and these tasks can execute if sufficient resources are available. However, if not enough resources are available, then Apollo uses correction mechanisms that continuously re-evaluate placements using up-to-date resource utilisation statistics. The Apollo scheduler starts a new task instance when it changes a task placement. In practice, this leads to inefficient resource utilisation because schedulers simultaneously run multiple task copies, and thus waste resources. Nonetheless, Apollo reduces task placement latency for non-interfering directly conflicting placements, but it does not efficiently handle indirectly conflicting placements without affecting task makespan and performance.

Distributed schedulers. Today's challenging workloads include short-running interactive tasks that complete within seconds (§2.1). Future workloads are likely to be even more challenging because more and more applications expect fast responses from infrastructure systems. This will cause task makespan to decrease and task throughput to increase. Such workloads require cluster schedulers to have: *(i)* low placement latency, and *(ii)* high placement throughput.

The *distributed scheduler architecture* is designed to meet the above-mentioned requirements. In this architecture, schedulers are fully distributed: they do not share state and do not require coordination [OWZ+13; RKK+16] (see Figure 2.12d). Moreover, distributed schedulers deliberately use simple algorithms amenable to fast parallel placements even at scale. They make placements using randomly sampled and gossiped information [OWZ+13; DDD+16]. However, distributed schedulers achieve poorer placement quality than other types of schedulers (e.g., monolithic, centralised schedulers) because each scheduler instance only has partial and often stale information about cluster state. Moreover, distributed schedulers sacrifice scheduling qualitybecause they do not support essential scheduling features such as co-location interference awareness (§2.3.1.2 and multi-dimensional resource fitting (§2.3.1.3).

In this dissertation, I show that there is no need to use distributed schedulers for current or even future workfloads. I describe several algorithms and techniques I developed that make Firmament, a min-cost flow centralised scheduler, place tasks with low placement latency at scale.

Hybrid schedulers. The *hybrid scheduler architecture* splits workloads across a centralised scheduler and one or more distributed schedulers (see Figure 2.12e). The architecture was developed to address distributed schedulers' poor placement quality and the perceived scalability limitations of monolithic, centralised schedulers.

Hawk [DDK+15] is a hybrid scheduler that uses statistics from previous task runs to assign tasks to either a centralised scheduler that places long-running tasks or to one of several schedulers that place short-running tasks. All Hawk's schedulers place tasks to machines that are statically partitioned into "slots". However, as I noted in § 2.3.1.3, slot-based schedulers do not choose high-quality placements because they assume that all tasks utilise the same amount of resources and interfere equally.

In contrast to shared-state schedulers, the Hawk hybrid scheduler queues tasks in machine-side queues when tasks do not fit on machines. However, tasks may queue after or wait for long-running tasks to complete, which may increase the makespan of short-running tasks by several orders of magnitude. In order to address this limitation, Hawk splits the cluster into a general pool of machines and a small dedicated pool of machines reserved for short-running tasks.

Mercury [KRC+15] is a hybrid cluster manager that offers two quality of service options: guaranteed and queable tasks. Mercury places guaranteed tasks with a centralised scheduler that has priority in case of conflict, and queable tasks with one of many low-priority distributed schedulers it runs. Finally, Eagle [DDD+16] is a cluster manager that extends Hawk with state gossiping techniques, and Yaq-d [RKK+16] is a scheduler that reorders tasks in machine-side queues to reduce latency for short-running tasks.

2.3.3 Cluster scheduling summary

Data center clusters have more machines and run more diverse types of tasks (§2.1). State-of-the-art cluster schedulers struggle to handle these workloads at scale. They either choose high-quality placements or offer low placement latency. As a result, resource utilisation in data centers is consistently below 20% [McK08; Liu12; DSK15]. In order to quickly place the workload and obtain high resource utilisation a cluster scheduler should:

1. consider hardware heterogeneity when scheduling tasks in order to reduce runtime or improve energy efficiency (§2.3.1.1);
2. avoid co-locating tasks that interfere on the same resource (§2.3.1.2);
3. consider task requirements of different resources and conduct multi-dimensional resource fitting (§2.3.1.3);
4. obtain high data locality in order to reduce network traffic and provide more predictable application performance (§2.3.1.4);
5. support placement constraints for tasks that have hardware preferences (§2.3.1.5);
6. estimate tasks' resource requirements and automatically adjust resource reservations when tasks do not fully utilise resource reservations (§2.3.1.6);
7. fairly share resources (§2.3.1.7);
8. scale to tens of thousands of machines at sub-second placement latency (§2.3.1.8).

In this dissertation, I show that with my extensions, the Firmament centralised cluster scheduler does not suffer from the limitations other monolithic, centralised schedulers do. In Chapter 5, I describe several techniques and algorithms that I developed to improve Firmament to choose high-quality placements with low task placement latency. Finally, in Chapter 6, I evaluate how Firmament compares to other schedulers. I focus on both placement latency and quality.

Chapter 3

Musketeer: an extensible workflow manager

Many data processing execution engines are designed to work well in specific cases such as large-scale string processing, iterative machine learning training, social network graph analytics; and most perform considerably *less* well when operating outside of their "comfort zone". Choosing the "best" data processing execution engine is difficult. It requires significant expert knowledge about the programming paradigm, design goals and implementation of the many available engines. Even with this knowledge, any move between execution engines requires time-consuming re-implementation of workflows. Furthermore, head-to-head comparisons and workflow runtime predictions are difficult because engines often make different assumptions and target different use cases. Developers therefore stick to their known, favourite front-end framework or back-end execution engine even if other, "better" engines offer superior performance or efficiency gains.

In Section 2.2, I evaluated a range of contemporary data processing execution engines – Hadoop, Spark, Naiad, PowerGraph, Metis and GraphChi – under controlled and comparable conditions. I found that *(i)* their performance varies widely depending on the high-level workflow; *(ii)* no single system always outperforms all others; and *(iii)* almost every system performs best under some circumstances (§2.2.3).

These differences are the result of execution engines developer making different optimisation trade-offs when addressing the following four challenges:

Input data size: execution engines are optimised for processing different amounts of data, and these optimisations are built-in the engines' architecture. For example, MapReduce is optimised for handling huge amounts of data that is stored on disk, Metis leverages parallelism within a single machine to process small data, and Spark covers the in-between spot of medium-sized data that fits into the memory of a cluster.

Data structure and skew: execution engines are tuned to handle data with a specific format because data structure and skew affect work distribution. For example, MapReduce is

suited for simple processing of bulk unstructured text, while PowerGraph is optimised for running computations on graphs with highly skewed power-law degree distributions.

Workflow type: different engines may be better suited depending on the workflow type (e.g., iterative, graph processing). Some engines are based on dataflow models that may not be well-suited to execute some workflows. For example, bipartite dataflow-based engines (e.g., Hadoop MapReduce) do not efficiently execute iterative computations, and BSP-based engines (e.g., Pregel) unnecessarily increase makespan of synchronisation-free graph workflows.

Engineering decisions: choices made while building execution engines affect performance. For example, for developers convenience, many batch processing execution engines are implemented in languages with automatic memory management. The workflows they execute allocate huge amounts of data and this often leads to increased makespan due to garbage collection pauses. Graph processing engines also make choices that affect makespan; PowerGraph optimises for running recurring computations on the same graph. It executes an expensive graph partitioning algorithm that increases first run's makespan. However, subsequent workflow runs complete faster.

To make matters worse, even if developers are familiar with the trade-offs the execution engines make, they cannot easily choose the most appropriate execution engines because they do not have information about key parameters at workflow implementation time (e.g., the size of generated intermediate data). I believe that nowadays, workflows do not complete as fast as they could because of the following four issues:

Execution engines are chosen statically, ahead of runtime: it is difficult to predict how much intermediate data a workflow will generate. Moreover, input data size can significantly change between two consecutive workflow runs because of temporary traffic spikes. Despite these input variations, workflows are statically implemented and executed in an execution engine decided upon ahead of runtime. This can lead to sub-optimal runs if the data volume or resource availability does not match implementation-time expectations.

Static, ahead of time workflow job boundaries: developers statically partition workflows into jobs and decide on which engines to execute them at implementation time. However, the best workflow partitioning depends on the expressivity of the dataflow models on which the available engines are based, on data structure and skew, and on the state of the cluster. This information is rarely available before workflow runtime. Thus, workflows should dynamically be partitioned into jobs at runtime.

Workflow migration difficulties: workflows must be manually re-written using new front-end frameworks or engine-specific interfaces upon the introduction of new execution engines. This requires significant engineering effort; hence discourages adoption of new execution engines, despite them offering performance gains or reduced resource utilisation.

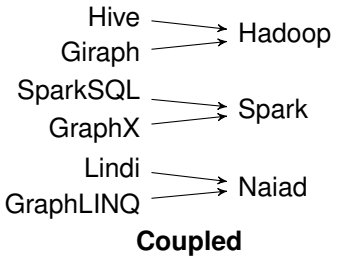


Figure 3.1: Front-end frameworks are coupled to a single back-end execution engine.

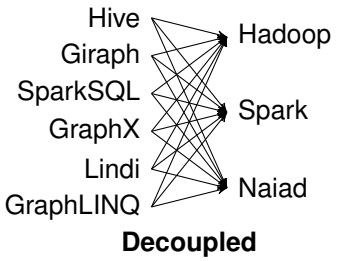


Figure 3.2: Decoupling front-end frameworks and back-end execution engines increases flexibility.

Job-level scheduling: workflows can comprise of several jobs that have dependencies among them. However, execution engines and cluster managers schedule workflows on a per-job basis, and do not take into account job dependencies. This can lead to unnecessary bottlenecks, resource waste and longer makespans [GKR+16].

The underlying reason for of which the above-mentioned issues arise is that developers are being forced to make decisions at implementation-time on which front-end framework and back-end execution engine to use, without having enough information. These decisions cannot be easily changed later because each front-end is tightly coupled to a single back-end execution engine (see Figure 3.1). In this chapter, I argue that *users should, in principle, be able to execute their high-level workflow on any data processing system*. To show that this is achievable in practice, I developed Musketeer, an extensible workflow manager that decouples front-end frameworks from back-end execution engines. It offers users the possibility of expressing workflows using a front-end without being locked into a back-end execution engine (see Figure 3.2).

In Section 3.1, I give an overview of Musketeer. Subsequently, in Section 3.2, I describe how workflows can be expressed using Musketeer-supported front-end frameworks. In Section 3.3, I introduce Musketeer's intermediate representation which it uses to decouple front-ends from back-ends. Following, in Section 3.4, I discuss the techniques Musketeer uses to generate efficient code to run workflows in back-end execution engines. Next, in Section 3.5, I describe how Musketeer automatically partitions and maps workflows to combinations of back-ends. Finally, in Section 3.6, I discuss Musketeer's limitations.

3.1 Musketeer overview

Musketeer is my proof-of-concept implementation of the decoupled design I advocate. In Musketeer, I break the execution of a data processing workflow into three high-level steps (Figure 3.2). First, a developer specifies her workflow using a *front-end framework*. Next, Musketeer translates this workflow specification into an *intermediate representation*. Third, Musketeer generates code for jobs from this representation and executes these jobs on one or more *back-end execution engines*.

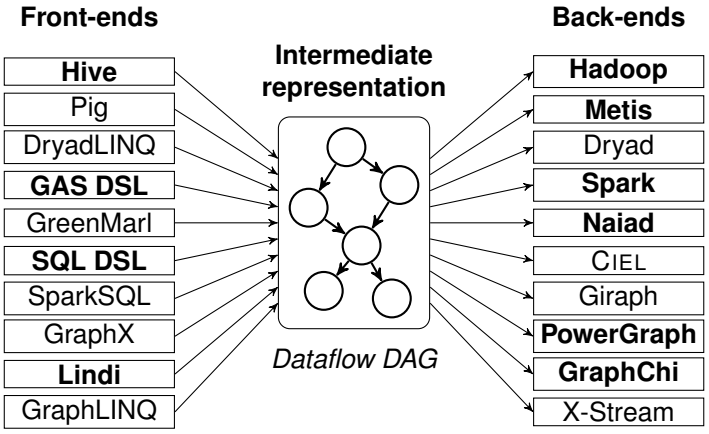


Figure 3.3: To decouple processing, Musketeer translates front-end workflow descriptions to a common intermediate representation from which it generates jobs for back-end execution engines. The Musketeer prototype supports the systems in **bold**.

I believe that by decoupling front-end frameworks from back-end execution engines, Musketeer offers four main benefits over prior solutions:

1. Developers write workflows only once for a back-end engine they choose, but can easily execute them on alternative back-ends.

2. Workflows independent partitions (i.e., jobs) can be executed on different back-end execution engines.

3. Developers do not have to micromanage workflows; they do not have to specify on which engines they should execute when input data sets change.

4. Existing workflows can easily be ported to new and more performance execution engines.

The above-mentioned benefits are possible because in Musketeer workflows proceed from specification to execution through seven modular phases (see Figure 3.4). I now give an overview of how these phases fit in the above-mentioned three high-level steps; following sections describe in detail the realisation of these phases in my Musketeer prototype.

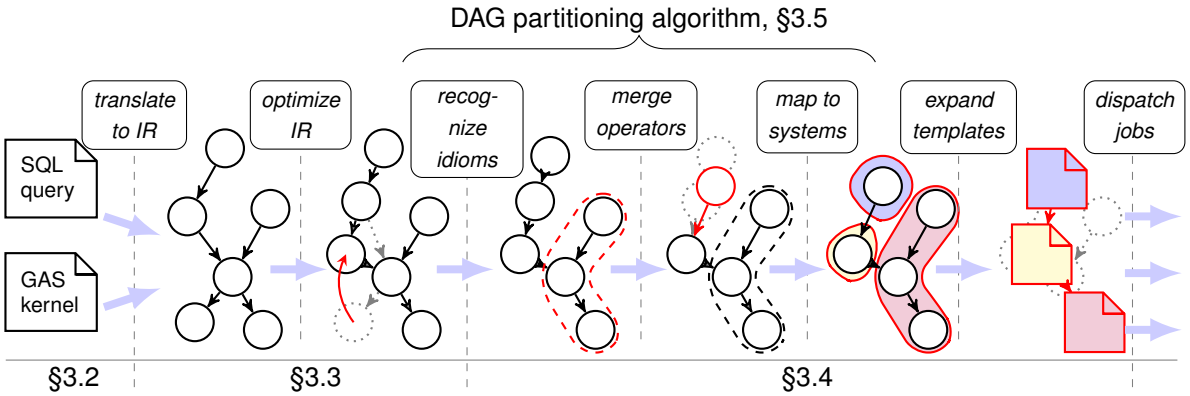


Figure 3.4: Phases of a Musketeer workflow execution. Dotted, grey operators show previous state; changes are in red; encircled operators represent jobs.

Front-ends. User-facing high-level abstractions for workflow expression ("frameworks") act as front-ends to Musketeer. Many such frameworks exist: SQL-like querying languages and vertex-centric graph abstractions are especially popular (see §2.2.2). While designing Musketeer's architecture, I assumed that users write their workflows for such front-end frameworks.

Musketeer currently supports four front-end frameworks: Hive, Lindi, SQL-like DSL and a gather, apply and scatter DSL. Musketeer integrates front-ends by either parsing user inputs directly, or by offering an API-compatible shim for each front-end. For example, Musketeer directly parses HiveQL queries and translates them into the common intermediate representation, whereas it supports Lindi workflows using an API-compatible Lindi reimplementation.

In Section 3.2, I present in detail how users can express workflows, describe the front-end frameworks Musketeer supports and explain how Musketeer integrates them.

Intermediate representation. Front-end frameworks can be decoupled from back-end execution engines in one of the following two ways: *(i)* by implementing a workflow translation shim layer for each front-end and back-end pair or, *(ii)* by developing an intermediate representation into which to translate front-end workflow specifications and from which to generate job code for back-end execution engines. In Musketeer, I chose the latter approach because it only requires each system to be translated to or from the intermediate representation. Whereas, the former approach would require Musketeer to provide many translation shim layers (i.e., for each front-end and back-end pair). The number of shim layers would quickly get out of hand because it would grow quadratically in the number of systems supported.

Moreover, the approach I chose is extensible: to add new front-end frameworks developers must only provide translation logic from framework constructs to the intermediate representation. Similarly, developers only must provide appropriate back-end job code templates and code generation logic to extend Musketeer to support new back-end execution engines. Finally, the intermediate representation also offers opportunities for sharing workflow optimisations across front-end frameworks and back-end execution engines. This approach is similar to the one taken

by the LLVM modular compiler framework [LA04], albeit in a different domain.

Ideally, in the future, all available front-end frameworks and back-end execution engines will agree on a single common intermediate representation (IR). In Section 3.3, I put forward an initial version of what I think this intermediate representation should look like. The IR must simultaneously: *(i)* be sufficiently expressive to support a broad range of workflows, and *(ii)* maintain enough workflow information such that Musketeer can optimise and generate from the IR back-end job code that is competitive with an optimised hand-written workflow implementation.

Musketeer's intermediate representation is a dynamic directed acyclic graph (DAG) of dataflow operators, with edges corresponding to input-output dependencies between operators. The operators are loosely based on Codd's relational algebra [Cod70]. This IR abstraction is general: it supports specific operator types based on relational algebra and general user-defined functions (UDFs). The IR supports iterative computations by dynamically expanding the DAG (as in CIEL [MSS+11] and Pydron [MAA+14]), and thus it does not have acyclic dataflow model's main limitation: inability to natively execute iterative computations. Moreover, the IR can also be extended with new operators if they are required to be able to perform new end-to-end optimisations.

Musketeer's intermediate representation is also well-suited for performing query optimisations which have been extensively studied by the database community [KD98; AH00; BBD05]. Equivalent techniques are already used in few front-ends (e.g., Spark SQL [AXL+15], FlumeJava [CRP+10]) or implemented in back-ends (e.g., Dryad's workflows are externally optimised by Optimus [KIY13]). Musketeer applies similar query optimisations on its intermediate representation for purposes such as reducing intermediate data sizes where possible. In Musketeer, these optimisations only have to be implemented once on the IR, whereas prior solutions require them to be implemented in each front-end and back-end. In Subsection 3.3.1, I describe several query optimisations that are currently supported by Musketeer.

Back-ends. Finally, Musketeer must generate code for specific distributed back-end execution engines. In Section 3.4, I describe in detail how Musketeer does it. A naïve approach would simply generate a job for each IR operator, but this approach suffers from high per-operator job startup costs, and fails to exploit opportunities for optimisation within execution engines (e.g., sharing data scans among operators, in-memory storage of intermediate data). Workflows typically benefit if they can be executed in as few independent jobs as possible. In Subsection 3.4.1, I describe how Musketeer is able to *merge operators* and generate code for them in a single job.

However, even if Musketeer is able to merge operators, there are some execution engines that have limited expressivity (e.g., MapReduce cannot compute a three-way join on different columns in a single job). Therefore, the IR dataflow DAG may still have to be partitioned into multiple jobs. Many valid partitioning options exist, depending on the workflow and the execution engines available. In Section 3.5, I show that exploring this space is an instance of

an NP-hard problem (k-way graph partitioning), and introduce a heuristic to solve it efficiently for large DAGs. Finally, given a suitable partitioning, Musketeer *generates* jobs for the chosen execution engines and *dispatches* them for execution.

3.2 Expressing workflows

Distributed execution engines simplify data processing on clusters of commodity machines by shielding developers from the intricacies of writing parallel, fault-tolerant code, and from the challenges of partitioning data and computation scheduling over thousands of machines. Nevertheless, they require developers to express computations in terms of low-level primitives, such as `map` and `reduce` functions [DG08] or message-passing vertices [MMI+13]. As a result, higher-level front-end frameworks that expose more convenient abstractions are commonly built on top (see §2.2.2). For example, Hive [TSJ+09] and Pig [ORS+08] front-ends present to developers SQL-like querying interfaces over the Hadoop MapReduce execution engine; Spark SQL and GraphX [GXD+14] offer SQL primitives and a vertex-centric interface over Spark [ZCD+12]; and Lindi and GraphLINQ [McS14] provide the same over Naiad [MMI+13].

In designing Musketeer, I rely on developers' preference to express workflows using these high-level front-end frameworks rather than to tediously implement workflows using the low-level interfaces provided by execution engines. This is common in industry: according to a recent study [CAK12], up to 80% of workflows running in production clusters are expressed using front-end frameworks such as Pig [ORS+08], Hive [TSJ+09], DryadLINQ [YIF+08] or Shark [XRZ+13].

Musketeer currently supports four front-end frameworks (Hive, Lindi, a custom SQL-like DSL with support for iterations, and a graph-oriented "Gather-Apply-Scatter" DSL). These front-ends can be grouped into three types: *(i)* SQL-like query languages (Hive, SQL-like DSL), *(ii)* language integrated queries (Lindi), and *(iii)* vertex-centric interfaces ("Gather-Apply-Scatter" DSL). Musketeer decouples all these types of front-ends from the back-ends they have been developed for by translating workflows to its common intermediate representation. I considered the following three approaches for achieving workflow translation:

Query parsing: Several front-end frameworks (e.g., Hive, Pig) use the ANTLR parser generator to transform each workflow query into an abstract syntax tree out of which they generate MapReduce jobs. One way to decouple these front-ends from back-ends is to translate the abstract syntax tree into Musketeer's IR rather than generate code. An alternative is to implement a parser that directly parses workflow queries and translates them to the IR.

Automatic translation of workflow code: Workflows implemented in imperative and functional languages could be automatically translated to Musketeer's IR using symbolic ex-

ecution. Prior systems exist that do this for single back-ends. For example, HadoopToSQL [IZ10] uses symbolic execution to derive preconditions and postconditions for MapReduce workflows, and generates SQL queries out of these. Similarly, QBS (Query By Synthesis) [CSM13] automatically translates imperative code into SQL queries. In contrast to HadoopToSQL, QBS uses constraint-based synthesis to extract queries from code invariants and postconditions expressed in relational algebra.

API-compatible shim: Some interfaces for expressing workflows (e.g., vertex-centric, and gather, apply and scatter) require developers to implement several functions. However, these functions cannot automatically be translated to Musketeer's IR because they do not have one-to-one mappings to the IR operators. For example, automatic translation solutions cannot translate the *gather* function from a GAS interface because they cannot infer from invariants and preconditions that both a `JOIN` and a `GROUP BY` operator are required to express the function [IZ10; CSM13]. These workflows also cannot be easily parsed because they are implemented in low-level languages such as C++. However, to support such workflows in Musketeer, I could provide an API-compatible shim that offers a subset of the low-level language's features, but is still rich enough to express the workflows.

I now turn to describing in detail the front-ends Musketeer supports and how it translates them to its IR.

3.2.1 SQL-like data analytics queries

Front-end frameworks that provide query languages based on SQL are widely used to express relational queries on data of tabular structure. For example, in 2009, Facebook was storing and processing 2 PB of data using Hive [TSJ+09; HIV16]. All data analysis computations at Microsoft are written in a SCOPE, a SQL-based front-end, and are executed on clusters of tens of thousands of commodity machines [CJL+08; BEL+14]. Similarly, many data analytics workflows are implemented in SQL-like front-ends such as Tenzing [CLL+11] at Google and Pig [ORS+08] at Yahoo!.

I integrate Hive as one of Musketeer's front-ends in order to demonstrate Musketeer's suitability to execute and reduce the makespan of workflows implemented in the above-mentioned front-ends.

Hive. In the Hive front-end framework workflows are expressed using HiveQL, a SQL-like language that provides query operators such as `SELECT`, `PROJECT`, `JOIN`, `AGGREGATE` and `UNION`. HiveQL has a basic type system that supports tables with primitive types, and simple collections such as arrays and maps. In Listing 3.1, I show an example Hive analytics workflow that computes the most expensive property on each street for a real-estate data set.

```
1 SELECT id, street, town FROM properties AS locs;
2 locs JOIN prices ON locs.id = prices.id AS id_price;
3 SELECT street, town, MAX(price) FROM id_price
4   GROUP BY street AND town AS street_price;
```

Listing 3.1: Hive query for the *max-property-price* workflow. Properties' locations are stored in the *properties* table and property prices are stored in the *prices* table.

My Musketeer prototype supports queries written in HiveQL. It directly parses most HiveQL operators and translates them to the intermediate representation. However, Musketeer does not currently support tables with non-primitive types, nor is able to execute Hive user-defined functions (UDFs). These limitations are not fundamental; they could be addressed with significant engineering effort.

In hindsight, I believe there is an easier way of integrating HiveQL. The query language uses the ANTLR parser generator to transform each query into an abstract syntax tree. Rather than implementing my own parser, I should have used the ANTLR parser to translate from the abstract syntax trees to Musketeer's intermediate representation. This approach would have not required me to handle Hive's idiosyncrasies.

Hive and the other above-mentioned SQL-like front-end frameworks are well-suited to express and run non-iterative batch workflows. However, many data analytics computations are iterative and only complete when a data-dependent convergence criterion is met. For example, the *k*-means algorithm iterates until no points change clusters between iterations. Similarly, PageRank iterates until the differences for all pages between previous page rank value and current rank is smaller than an error factor. Such iterative workflows cannot be expressed in the SQL-like front-ends because they model workflows as directed acyclic graphs of SQL-like operators. In practice, developers express workflows' loop bodies using these front-ends and write driver programs to check convergence criteria and dispatch workflows (§2.2.1).

BEER. I implemented my own SQL-based domain-specific language that can express iterative workflows. Like other SQL-like query front-ends, BEER operates on typed tables, which can be accessed by columns. BEER is based on relational algebra and offers operators such as `SELECT`, `PROJECT`, `JOIN`, `UNION` and `DIFFERENCE`. It also supports aggregate operators `MAX`, `MIN`, `COUNT`, and arithmetic operators `MUL`, `DIV`, `SUB` and `SUM`. In contrast to other SQL-like front-ends, BEER also provides a `WHILE` operator that can be used to implement iterative workflows with data-dependent convergence criteria. In Listing 3.2, I show a BEER implementation of the PageRank iterative workflow that uses the `WHILE` operator.

Workflows often contain user-defined functions that are used to express specialised data processing tasks [ORS+08; KPX+11]. Thus, it is critical to support UDFs in Musketeer. BEER has a `UDF` construct that can be used to specify code paths to UDFs. The code to which these paths point to is later included in the back-end specific code Musketeer generates. This ap-

```
CREATE RELATION edges WITH COLUMNS (INTEGER, INTEGER),
// Relation storing (page_id, rank) pairs.
CREATE RELATION page_rank WITH COLUMNS (INTEGER, DOUBLE),
// Relation storing current iteration number.
CREATE RELATION iter WITH COLUMNS (INTEGER),
// Computes number of outgoing edges for each graph node.
COUNT [edges_1] FROM (edges) GROUP BY [edges_0] AS node_cnt,
// Pre-computes 3-tuples of (src_node, dst_node, src_outgoing_degree).
(edges) JOIN (node_cnt) ON edges_0 AND node_cnt_0 AS edges_cnt,
WHILE [(iter_0 < 20)] (
  // Joins the pre-computed 3-tuples with the current page ranks.
  (edges_cnt) JOIN (page_rank) ON edges_cnt_0 AND page_rank_0 AS edges_pr,
  DIV [edges_pr_3, edges_pr_2] FROM (edges_pr) AS rank_cnt,
  PROJECT [rank_cnt_1, rank_cnt_3] FROM (rank_cnt) AS links,
  AGG [links_1, +] FROM (links) GROUP BY [links_0] AS page_rank1,
  MUL [0.85, page_rank1_1] FROM (page_rank1) AS page_rank2,
  // Updates each node's page rank value.
  SUM [0.15, page_rank2_1] FROM (page_rank2) AS page_rank,
  // Increases iteration counter.
  SUM [iter_0,1] FROM (iter) AS iter)
```

Listing 3.2: BEER DSL code for PageRank. The workflow conducts 20 PageRank iterations in which it updates each vertex's page rank value stored in *page_rank* table.

proach works well for workflows that run on back-ends that can execute or interface with code implemented in the users' programming language of choice. However, this approach restricts the back-end choice because some workflows can only be executed in less efficient back-ends that support the programming language in which the UDFs are implemented.

Expert developers understand well the intricacies of back-end execution engines and can sometimes make counterintuitive, but good decisions when developing their workflows. My DSL language offers a mechanism to avoid translating parts of workflows to Musketeer's intermediate representation. Expert developers can use a `BLACK-BOX` operator to provide complete job implementations for key parts of their workflows. Musketeer executes these jobs on the back-end execution engines they were implemented for.

In my BEER implementation, I use the ANTLR parser generator to create a parser out of the language's grammar. Following, I use this parser to transform workflow queries into abstract syntax trees. Finally, I map these trees of BEER relational algebra operators to Musketeer's intermediate representation; most operators have directly corresponding Musketeer IR operators (§3.3).

3.2.2 Language integrated queries

Often workflows are fully integrated in applications' logic. In such cases, it is appealing to use a language integrated query front-end such DryadLINQ [YIF+08] or Lindi [MMI+13]. These

front-ends add native SQL-like querying expressions to common programming languages.

Lindi. Is a C# language extension that can be used to express workflows as sequential programs of LINQ operators. These LINQ operators run user-provided code and conduct transformations of data sets. Like many other front-ends, Lindi can currently only run workflows on a single back-end, the Naiad execution engine. Internally, Lindi implements a Naiad vertex for each LINQ operator it offers.

Supporting the complete Lindi front-end in Musketeer is challenging because Lindi's operators execute user defined functions. These functions could be applied by Musketeer using the same mechanism as in BEER, but this would restrict the back-ends on which Lindi workflows can run. The UDFs would have to execute in back-ends that support the programming languages in which the workflows are implemented. However, this limitation could be addressed by: *(i)* automatically translating UDFs to different languages using source-to-source compilers or, *(ii)* implementing a Lindi API-compatible shim that offers a subset of C# features, but has enough features to express the workflows. I chose the latter approach in Musketeer because state-of-the-art source-to-source compilers can only translate limited language subsets [Pla13]. I implemented a Lindi API-compatible shim in C++, which I also extended with an operator that can be used for iterative workflows. In Listing 3.3, I show the Lindi-like API supported by Musketeer.

3.2.3 Graph data analysis

Domain-specific front-end frameworks for expressing graph computations are popular: Pregel and Giraph run computations on MapReduce [MAB+10], GraphLINQ offers a graph-specific API for the Naiad execution engine [McS14], and GreenMarl DSL emits code for few multi-threaded and distributed runtimes [HCS+12]. Many of these front-ends provide a vertex-centric API in which users provide code that is concurrently instantiated for each vertex [MAB+10; KBG12]. This vertex-centric programming pattern is generalised by the **G**ather, **A**pply and **S**catter (GAS) model in PowerGraph [GLG+12]. In this paradigm, data are first gathered from neighbouring nodes, then vertex state is updated and, finally, the new state is disseminated (scattered) to the neighbours.

In Musketeer, graph processing workflows can be expressed using the BEER DSL. However, BEER requires workflows to be modelled using relational algebra operators, which can often be non-intuitive (see Listing 3.2). To address this, I developed a domain-specific front-end framework that combines the GAS programming model with my BEER DSL. To express graph workflows in my graph "Gather-Apply-Scatter" DSL, users must define the three GAS steps. For each step they must use BEER relational operators or user-defined functions (UDFs). In Listing 3.4, I show the implementation of the PageRank workflow in my GAS front-end framework.

```
OperatorNode Select(OperatorNode op_node, vector<Column> select_columns,
                    ConditionTree cond_tree, string rel_output_name);

OperatorNode Where(OperatorNode op_node, ConditionTree cond_tree, string rel_output_name);

OperatorNode SelectMany(OperatorNode op_node, ConditionTree cond_tree, string rel_output_name);

OperatorNode Concat(OperatorNode left_op_node, string rel_output_name, OperatorNode right_op_node);

OperatorNode GroupBy(OperatorNode op_node, vector<Column>& grouped_columns, GroupByType group_reducer,
                     Column group_by_column, string rel_output_name);

OperatorNode Join(OperatorNode left_op_node, string rel_output_name, OperatorNode right_other_op_node,
                  vector<Column> left_join_cols, vector<Column> right_join_cols);

OperatorNode Distinct(OperatorNode op_node, string rel_output_name);

OperatorNode Union(OperatorNode left_op_node, string rel_output_name, OperatorNode right_op_node);

OperatorNode Intersect(OperatorNode left_op_node, string rel_output_name, OperatorNode right_op_node);

OperatorNode Except(OperatorNode left_op_node, string rel_output_name, OperatorNode right_op_node);

OperatorNode Count(OperatorNode op_node, Column cnt_column, string rel_output_name);

OperatorNode Min(OperatorNode op_node, Column group_by_column,
                 vector<Column>& min_columns, string rel_output_name);

OperatorNode Max(OperatorNode op_node, Column group_by_column,
                 vector<Column> max_columns, string rel_output_name);

// Extension for iterative computations.
OperatorNode Iterate(OperatorNode op_node, ConditionTree cond_tree, string rel_output_name);
```

Listing 3.3: Musketeer's Lindi-compatible interface.

```
GATHER = {
  SUM (vertex_value)
}
APPLY = {
  MUL [vertex_value, 0.85]
  SUM [vertex_value, 0.15]
}
SCATTER = {
  DIV [vertex_value, vertex_degree]
}
ITERATION_STOP = (iteration < 20)
ITERATION = {
  SUM [iteration, 1])
}
```

Listing 3.4: Gather-Apply-Scatter DSL code for PageRank.

The front-ends I discussed previously (i.e., HiveQL and my BEER DSL) have directly corresponding operators in Musketeer's IR, but this is not the case for my GAS DSL. The DSL requires both syntactic translation and transformation from its gather, apply, and scatter paradigm to the IR operator dataflow DAG. Musketeer achieves this by modelling message exchanges in each iteration (i.e., scatter step) as a `join` between a relation storing vertices and a relation storing edges. Moreover, it models gather steps by appending a `group-by` clause to each operator used in these steps. Musketeer's techniques are similar with the ones used by Pregelix to offer Pregel-like semantics on top of Hyracks, a general purpose shared-nothing dataflow engine [BBJ+14].

3.2.4 Other types of workflows

In addition to SQL-like queries, integrated language queries and GAS graph computations, Musketeer could also support other types of workflows. These workflows could directly be translated to Musketeer's IR, if they are expressed using abstractions that map one-to-one to the IR operators. Abstractions for which no IR operator exists can be mapped to several operators, to user-defined functions (UDF), or to a "native" back-end via the "black box" operator. I discuss extending Musketeer with other front-ends in §3.6.

3.3 Intermediate representation

The common intermediate representation (IR) lays at Musketeer's core. Workflows implemented in different front-ends are translated to it and, back-end execution engine code is generated from it. In order for the intermediate representation to successfully decouple front-ends from back-ends it must have the following three properties:

Expressivity: the IR must be general enough to express all the different types of workflows supported by front-end frameworks (i.e., batch, iterative and graph processing).

Efficiency: the IR must not lose information associated with workflows, whose absence may increase makespan. For example, for a graph processing workflow, Musketeer must be able to infer from the intermediate representation that the workflow is a graph computation.

Straightforward mappings to back-ends: the IR must not be too high-level (e.g., operators for common algorithms) to miss on optimisation opportunities, but it also must not be too low-level (e.g., LLVM bytecode) to not have direct mapping to operators supported by back-ends.

Common intermediate representations are used in other problem domains as well. For example, the LLVM [LA04] compiler framework compiles a range of programming languages (e.g., C, C++, Haskell) into a light-weight and low-level IR. LLVM's IR is typed and expressive enough to be used to efficiently conduct compiler optimisations, transformations and analysis. These optimisations must only be implemented once because the LLVM IR abstracts away the details of target machines. LLVM generates code for different instructions sets from its IR.

Similarly to Musketeer, Weld aims to reduce workflow makespan, but it focuses on single-machine code [PTS+17]. It uses compiler techniques such as loop fusion, loop tiling and vectorisation to generate efficient code. Like Musketeer, Weld [PTS+17] uses an IR to first capture computations from different front-ends. Following, it transforms the IR code into efficient machine code. Weld's IR comprises of parallel loops and builders with declarative data types that are used for computing results in parallel (e.g., computing a sum). The builders do not specify an implementation within the IR. Instead, Weld dynamically chooses between different implementations based on the hardware available and the data processed.

I considered similar approaches for Musketeer's IR, but I concluded that while LLVM and Weld's IRs are efficient and general enough to express the workflows implemented in front-ends, there are no straightforward ways to map workflows from these IRs to back-end execution engines' APIs. LLVM's IR bytecode encoding is too low-level to extract mappings from it. Even for a simple workflow that joins two data sets, Musketeer would have to statically analyse the IR bytecode to rediscover that the code is conducting a join and that the workflow should be executed in a join-specialised execution engine. Similarly, Weld's IR based on parallel loops and builders does not have clean and decidable mappings to back-ends. For example, it is not trivial to detect if a given Weld IR code is expressing a graph processing workflow.

By contrast, in my Musketeer prototype I developed a higher-level IR that combines the acyclic and the dynamic dataflow models. All back-end execution engines targeted by Musketeer are built on one of the following four dataflow models: bipartite, acyclic, dynamic and timely. Therefore, Musketeer's IR is well-suited to map workflows to these back-ends. I first considered using the bipartite dataflow model for the IR, but the model is not expressive enough

to represent workflows that shuffle data more than once (i.e., workflows that contain two or more join operators). Following, I considered the acyclic dataflow model which can express the abovementioned workflows, but the model cannot represent workflows that require data-depedent or unbounded iterations. This limitation is addressed in the dynamic dataflow model in which dataflow operators can spawn additional operators at runtime. In this model, iterative workflows can decide at the end of each iteration if operators should be spawned for an additional iteration.

Musketeer's IR is an acyclic directed graph of dataflow operators, but like in the dynamic dataflow model, Musketeer can dynamically extend the graph for iterative workflows. It achieves this with a special `WHILE` operator that dynamically spawns operators for its body at runtime. Unlike in the dynamic dataflow model, in Musketeer no other operators can dynamically spawn operators. Nonetheless, Musketeer's IR is expressive enough to support most workflows [IBY+07; ZCD+12] and amenable to analysis and optimisation [KIY13; KAA+13].

An alternative approach is to use the timely dataflow model for Musketeer's IR. Timely dataflow represents workflows as directed cyclic graphs in which edges carry both data records and logical timestamps. The timestamps are used to pass iteration data and workflow progress information that is used to check completion conditions of data-dependent iterative workflows. Timely dataflow is an expressive model that can even represent computations (e.g., incremental and differential computations) that are too complex to be expressed in other models. Musketeer does not use timely dataflow for its IR because the two were developed concurrently. However, in the future, I could change Musketeer's dynamic acyclic operator DAG IR with an IR based on timely dataflow.

The initial set of operators I use for Musketeer's IR is loosely based on Codd's relational algebra [Cod70] and covers the most common operations from industry workflows [CAK12]. Musketeer's relational algebra-based set of operators includes `SELECT`, `PROJECT`, `UNION`, `INTERSECT`, `JOIN` and `DIFFERENCE`, aggregators (`AGG`, `GROUP BY`), column-level algebraic operations (`SUM`, `SUB`, `DIV`, `MUL`), and extremes (`MAX`, `MIN`). However, standard relational algebra is not expressive enough to meet my goals because it cannot represent queries that compute the transitive closure of a relation [Mur11]. Nonetheless, this limitation can be addressed by extending the set of operators with a fixed point operator that enables recursive computation [AU79]. I extend the IR with a `WHILE` operator that can be used to express dynamic data-dependent iterations. The operator dynamically extends the IR DAG based on the output of the operators used in `WHILE`'s condition. As shown by Murray [Mur11, §3.3.3], a DAG with this facility is sufficient to achieve Turing completeness and it can express all while-programs (though not necessarily efficiently). Finally, I also include a special `UDF` operator in Musketeer's IR to support user-defined functions.

Musketeer's IR set of operators is, in my experience, already sufficient to model many widely-used data processing paradigms. For example, MapReduce workflows can be directly modelled with the `MAP`, `GROUP BY` and `AGG` operators. Similarly, complex graph workflows can be

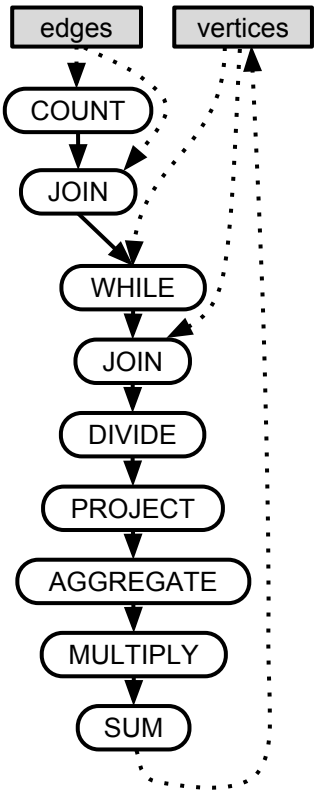


Figure 3.5: PageRank workflow represented in Musketeer's intermediate representation.

mapped to a specific `JOIN`, `MAP`, `GROUP BY` pattern, as shown by GraphX [GXD+14] and Pregelix [BBJ+14]. In Figure 3.5, I show how such a graph processing workflow (PageRank) is represented in Musketeer's IR. Even if Musketeer's IR is simple, not all its operators are used by all front-ends, and not all back-ends support all its operators. Nonetheless, the IR can be further extended with operators if future workflows require them.

3.3.1 Optimising workflows

Many front-end frameworks optimise workflows before execution. Pig [ORS+08], Hive [TSJ+09], Shark [XRZ+13] and SparkSQL use rewriting rules to optimise relational queries. Similarly, FlumeJava [CRP+10], Optimus [KIY13] and RoPE [AKB+12a] apply optimisations to operator DAGs. Yet, each optimisation is implemented independently for each front-end framework.

One of the advantages of decoupling front-ends from back-ends is that optimisations can be implemented and applied only once on the intermediate representation. This is leveraged in the LLVM modular compiler framework in which many optimisation passes are made over the IR to produce programs that run faster [LA04]. Musketeer can likewise provide benefits to all supported front-ends – and future ones – by applying optimisations on the intermediate representation.

My Musketeer prototype performs a small set of standard query rewriting optimisations on the IR, while preserving workflow semantics. Most of these re-order operators (e.g., bring

selective ones closer to the start of the workflow and push generative operators to the end) in order to reduce the amount of data processed by subsequent operators. For example, Musketeer moves selective operators such as `PROJECT`, `SELECT` and `INTERSECT` before non-selective operators they commute with (e.g., `AGG`).

More advanced optimisations could be applied on Musketeer's IR. These optimisation could prune unused columns of intermediate operators whose output must be written on disk or in memory. Similarly, the optimisations could move highly selective `JOIN` operators closer to the start of workflows, and push `JOIN` operators that generate large amounts of data closer to the end of workflows. However, I do not implement these optimisations because large-scale data processing query rewriting is not the focus of my work.

In comparison to front-end frameworks or database systems, it is more difficult to predict the effect of query rewriting rules in Musketeer. Optimisation rules that are guaranteed to reduce makespan in other systems might not always have the same effect because Musketeer dynamically maps workflows to back-ends at runtime. In particular, optimisations may increase the minimum number of jobs required to run a workflow and thus potentially increase makespan. In Section 3.6, I discuss how Musketeer could address this limitation.

3.4 Code generation

Musketeer decouples front-end frameworks from back-end execution engines by mapping workflows to its intermediate representation. In order for this approach to work well in practice, Musketeer must generate efficient code from its intermediate representation for back-end execution engines. This is a challenging undertaking because back-ends offer a diverse set of interfaces for expressing workflows. For example, Musketeer must generate simple stateless *map* and *reduce* functions (for Hadoop MapReduce), or implementations for complex stateful vertices (for Naiad).

Moreover, back-ends often require Musketeer to provide workflow code in particular programming languages. For example, Metis expects workflows to be implemented in C++ and to manage their own memory, whereas Naiad expects workflows implemented in C# and memory management is provided by the Common Language Runtime (CLR).

To sum up, in order to efficiently integrate back-ends, Musketeer's code generation mechanism must meet the following three requirements:

- generate workflow code that is competitive with hand-written optimised implementations for each back-end;
- generate code for a range of back-ends which provide diverse APIs and are implemented in programming languages with different syntax;

- offer an intuitive and easy-to-use interface for integrating new back-end execution engines.

I discuss three approaches to implement Musketeer's code generation mechanism. First, Musketeer could generate code in a low-level language (e.g., C++) for each IR operator. Following, it would either directly execute the code in back-ends that support the chosen programming language, or plug in the code in jobs that execute on back-ends supporting other programming languages using foreign function interfaces (FFI). Musketeer would have to generate code from the IR to only a single language, but it could find it challenging to transform workflows that use complex data types. Moreover, FFIs may be expensive to use because most back-ends create an object for each input record. These objects would have to be transformed to FFI-compatible objects. Bacila demonstrated that foreign function interfaces can add up to 10% overhead to Musketeer's generated workflow code compared to baselines without FFIs [Bac15].

In an alternative approach, Musketeer could use source-to-source compilers to translate the code it generates to different programming languages. Source-to-source compilers would add a small per-operator overhead, but would not add overhead to each input record. However, source-to-source compilers can only translate limited language subsets [Pla13], and I would have to build shim layers to interface between back-end specific objects and translated code for some back-ends.

The approach I choose in Musketeer is to use code templates for each IR operator and back-end pair. Conceptually, Musketeer instantiates and concatenates code templates to produce executable workflow code. This approach has the disadvantage of requiring templates to be implemented for each operator when a new back-end is integrated. However, templates only need to be implemented once and can be used many times to generate efficient code in the supported programming languages and using the interfaces provided by back-ends.

ToDo #1: Should I show the interface that must be implemented to integrate frameworks?

In practice, however, optimisations are required to make the performance of the generated code competitive with hand-written baselines. Musketeer uses traditional database optimisations (e.g., *sharing data scans* and *operator merging*), combined with compiler techniques (e.g., *idiom recognition* and *type inference*) to improve code generation. I explain these optimisations with respect to the *max-property-price* Hive workflow (see Listing 3.1) and the BEER PageRank workflow (see Listing 3.2).

Musketeer can currently generate efficient code for seven back-end execution systems: Hadoop MapReduce, Spark, Naiad, PowerGraph, GraphChi, Metis and serial C code by using templates and the above-mentioned optimisations. For the time being, I assume that the user explicitly specifies which back-end execution engines to use; in §3.5, I show how Musketeer can automatically map workflows to back-ends.

MapReduce job class	*Operators*	*Shuffle key*
map-only	`PROJECT`, `SELECT`, `SUM`, `SUB`, `DIV`, `MUL`, `UNION`	–
no-op-map and reduce	`INTERSECT`, `DIFFERENCE`	entire input row
aggregating-map and reduce	`AGG`, `COUNT`, `MAX`, `MIN`	`GROUP BY` columns, if present
map and reduce	`JOIN`	values in columns joined on

Table 3.1: Musketeer's IR operators fall under different MapReduce job classes. Some (e.g., `PROJECT`) do not need the reduce phase, while others (e.g., `INTERSECT`) only need the reduce phase.

3.4.1 Merging operators

Up to now, I described Musketeer as a workflow manager that generates code and executes each operator in a separate job. However, this approach has prohibitive makespans; regardless of back-end, each operator incurs a significant job start-up cost: the job must load input data, build up internal data structures, and coordinate parallel workers. This cost is especially high in general-purpose back-ends that store intermediate data in-memory (e.g., Spark, Naiad) because they can run most workflows in a single job, and thus do not usually incur per operator data loading costs.

Musketeer therefore *merges* operators in order to reduce the number of jobs executed. Ideally, Musketeer merges all operators within workflow in a single job, but in practice it may not be able to do so. The types of operators Musketeer can merge depends on the available back-end execution engines and the dataflow models these are based on. For example, specialised graph processing back-ends based on the bulk synchronous parallel model (e.g., Pregel) or the asynchronous model (e.g., PowerGraph) only support few operator idioms (see 3.4.2).

Similarly, back-ends based on the restrictive bipartite dataflow model (e.g., Hadoop MapReduce) may execute workflows using multiple jobs. Hadoop MapReduce jobs comprise of two explicit phases (*map* and *reduce*) and an implicit *shuffle* phase. The *shuffle* phase sorts key-value pairs returned from the *map* phase, groups them by key, and sends them to the *reduce* phase. As a result, workflows that contain several operators that each require a *shuffle* cannot be executed in a single Hadoop MapReduce job unless the *shuffle* phases can be composed (i.e., if they use the same key to sort the key-balue pairs).

I divide the Musketeer IR operators into four classes (see Table 3.1) depending on the phases they use:

1. **map-only** operators (e.g., `PROJECT`, `SELECT`, `DIV`, `MUL`, `SUB`, `SUM`, `UNION`) do not require the shuffle and reduce phases. Musketeer can merge two or more map-only operators by concatenating the operators' map phase code. Moreover, Musketeer can merge

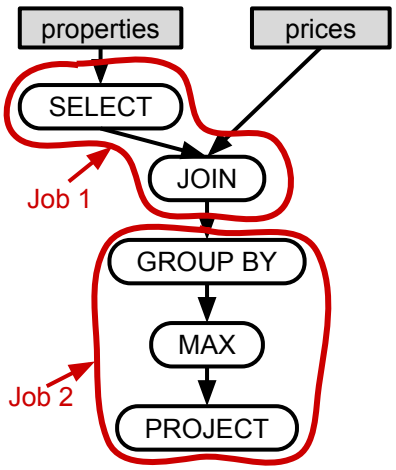


Figure 3.6: Max-property-price workflow represented in Musketeer's IR. Musketeer generates code for two jobs when users desire to run the workflow on Hadoop MapReduce.

map-only operators with any other class of operators by either concatenating their map phase code into the map or reduce phases of other operators.

2. **no-op-map and reduce** operators (e.g., `INTERSECT`, `DIFFERENCE`) have no-op map phases, but use the shuffle phase to group all identical input rows, and to dispatch them to a reducer. Musketeer can merge two or more map-only and no-op-map reduce operators into a job. However, it cannot merge no-op-map and reduce operators with other classes of operators that require a shuffle phase.

3. **aggregating-map and reduce** operators (e.g., `AGG`, `COUNT`, `MAX`, `MIN`) use the map phase to locally aggregate rows within an input data partition, the shuffle phase to group the aggregated results, and the reduce phase to aggregate the data across input partitions. Musketeer can only merge these operators with map-only operators.

4. **map and reduce** operators (e.g., `JOIN`) use all the three MapReduce phases. They cannot be merged with any other operator that has a shuffle phase, unless both operators use the same key in the shuffle phase.

Musketeer uses the above-mentioned rules to merge operators when possible. In Figure 3.6, I show how Musketeer divides the *max-property-price* workflow into jobs if it were to run the workflow on Hadoop MapReduce. Lines 1–3 in Listing 3.1 (§3.2.1) result in a job that selects columns from the `properties` relation and joins the result with the `prices` relation using `id` as the key; and *(ii)* lines 4–5 group by a *different* key than the prior join, and thus they are grouped into a second job.

The Hadoop MapReduce is not a good back-end choice for the max-property-price workflow because of additional job start-up overheads, and costs to serialise data to disk at the end of the first job and deserialisation from disk at the start of the second job. By contrast, other general-purpose back-ends can merge all classes of operators (e.g., Naiad, Spark). In Listing 3.5, I show the Spark code Musketeer generates for the *max-property-price* workflow.

```
prop_locations =
  properties.map(property => (property.uid, property.street, property.town))
prop_prices = prop_locations
  .map(prop => (prop.uid, (prop.street, prop.town)))
  .join(prices)
  .map((key, (left_rel, right_rel)) => (key, left_rel, right_rel))
street_price = prop_prices
  .map(prp_price => ((prp_price.street, prp_price.town), prp_price.price)
  .reduceByKey((left, right) => Max(left, right))
```

Listing 3.5: Naïve Spark code for *max-property-price*. Four `map` phases are required because data structures must be transformed to match the expected input format of dependent phases.

To sum up, Musketeer merges operators to reduce reduce job start-up costs and to reduce data serialisation costs. Operator merging is necessary for good performance: in §4.3, I show that it reduces workflow makespan by 2–5×.

3.4.2 Idiom recognition

Some back-end execution engines are based on restrictive dataflow models and are specialised for a single workflow type. For example, GraphChi offers a vertex-centric computation interface, and PowerGraph uses the gather, apply, scatter (GAS) decomposition to run graph analysis workflows. Neither back-end can execute workflows that do not conduct graph computations. Musketeer must therefore detect workflows that cannot be executed on these back-ends and in other specialised back-ends. It must also recognise computational idioms in the IR operator DAG to discover workflows which are suitable for specialised back-ends. *Idiom recognition* is a technique used in parallelising compilers to detect computational idioms that allow performance improving transformations to be applied [PE95]. I use a similar approach to detect computation patterns in the IR operator DAG.

My Musketeer prototype detects vertex-centric graph-processing workflows represented in its IR, even when workflows are originally expressed using relational front-ends (e.g., BEER instead of the GAS DSL). The idiom Musketeer uses is a reverse variant of the idom GraphX and Pregelix use to translate graph workflows to dataflow operators [BBJ+14; GXD+14]. Musketeer looks for a combination of the `WHILE`, `JOIN` and `GROUP BY` operators: the body of the `WHILE` loop must contain a `JOIN` operator with two inputs (i.e., the vertices and the edges). The `JOIN` operator must be followed by a `GROUP BY` operator that groups data by the vertex column.

This structure maps to graph computation paradigms: the `JOIN` on the vertex column is equivalent to sending messages to neighbour vertices (vertex-centric model), or the "scatter" phase (GAS decomposition); the `GROUP BY` is equivalent to receiving messages, or the "gather"

```
1 locs =
2   properties.map(c => (c.uid, (c.street, c.town)))
3 id_price = locs
4   .join(prices)
5   .map((key, (l_rel, r_rel)) =>
6           ((l_rel.street, l_rel.town), r_rel.price))
7 street_price = id_price
8   .reduceByKey((left, right) => Max(left, right))
```

Listing 3.6: Optimized Spark code for *max-property-price*. Scan sharing and type inference reduce the `map`s to two.

step (GAS); and any other operators in the `WHILE` body are part of the per-vertex computation (vertex-centric model) or the "apply" step (GAS), and are equivalent to updating vertex state.

Musketeer may occasionally fail to detect graph workloads, and, consequently execute them in less efficient back-ends. For example, a *graph triangle counting* workflow that computes all 3-tuples of vertices that form a triangle may not be recognised. A vertex forms a triangle with two other vertices if it is their neighbour and the other vertices are connected by an edge as well. Musketeer's idiom recognition algorithm does not detect this graph workflow because it can be expressed in two ways: *(i)* as a graph workflow using Musketeer's GAS DSL and, *(ii)* as a batch workflow that joins a table twice (i.e. the edges table) and filters out the results (i.e., selects the triangles). In the second case, Musketeer fails to take advantage of the opportunity to run the computation in a specialised graph execution engine because the workflow does not contain a `WHILE` operator. While my simple idiom recognition technique does not catch every opportunity of using specialised graph processing back-ends, it does not yield any false positives. It is safe to apply even across complex workflows.

3.4.3 Sharing data scans

Musketeer merges operators, executes them as a single job, and thus eliminates job creation overheads where possible. However, this optimisation is not enough to generate code that is competitive with hand-written baselines. Workflows do not incur overhead costs of starting multiple jobs, but data scans are still conducted fore each operator in some back-ends (e.g., Spark, Naiad). For example, Musketeer translates the first `SELECT` and the `JOIN` operator from the *max-property-price* workflow into two Spark `map` transformations and a `join` (Listing 3.5, lines 3–6). The first `map` implements the first `SELECT` operator, while the second `map` establishes a key $\rightarrow \langle$tuple$\rangle$ mapping over which the `join` is conducted. Even though Spark holds the intermediate RDDs in memory, scanning over the data twice yields a significant performance penalty.

Musketeer reduces data scans if the operators or several of their steps can be composed into a single function. It applies the composed function to each input record in a single pass over

the input data. For example, Musketeer generates optimised Spark code (Listing 3.6) for the *max-property-price* workflow in which it composes the anonymous lambdas from the first two map transformations (Listing 3.5, lines 4 and 6) into a single lambda (Listing 3.6, lines 5–6). Thus, the generated code selects the required columns and prepares the relation for the `join` transformation with only one data scan. Musketeer's optimisation is not limited to Spark code, Musketeer automatically generates code that avoids redundant scans for different back-ends.

3.4.4 Look-ahead and type inference

Many execution engines (e.g., Spark and Naiad) expose a rich API for manipulating data types. For example, the `SELECT ... GROUP BY` clause in the *max-property-price* workflow (Listing 3.1, lines 4–5) can be implemented directly in Spark using a `reduceByKey` transformation. However, such API calls often require specific input data formats. In my example, Spark's `reduceByKey` requires the data to be represented as a set of ⟨key, value⟩ tuples. Unfortunately, the preceding `join` transformation outputs the data in a different format (*viz.* ⟨key, ⟨left_relation, right_relation⟩⟩). The naïve generated code for Spark is inefficient because it contains *two* `map` transformations, one to flatten the output of the `join` (Listing 3.5, line 6), and another to key the relation by a ⟨`town`, `street`⟩ tuple (line 8).

In order to avoid generating superfluous data transformations, Musketeer looks ahead and uses type inference to determine the input format of the operators that ingest another operator's output. With this optimisation, Musketeer expresses the two `map` transformations as a single transformation (Listing 3.6, lines 5–6). The optimisations I described in this section enable Musketeer to generate code that has comparable performance with optimised hand-written baselines.

3.5 DAG partitioning and automatic mapping

Many execution engines cannot run complex workflows in a single job (see §3.4.1). For example, execution engines based on the bipartite dataflow model (e.g., MapReduce) can execute only one `GROUP BY` operator per job, and vertex-centric engines (e.g., PowerGraph, GraphChi) are restricted to the idiom described in §3.4.2. Moreover, even general execution engines have built-in assumptions about the likely operating regime and the types of workflows they expect to run. Their implementation is optimised based on these assumptions (§2.2.3). For these reasons it is difficult to find the "best" combination of systems to run a workflow.

Workflow managers such as Pig optimise workflows and generate code for all jobs before runtime. However, the "best" back-end choice not only depends on the input, but on intermediate data size as well, which is not available before runtime. By contrast, Musketeer continuously updates data size statistitcs and dynamically decides at runtime which back-ends to use. Thus, Musketeer can decrease workflow makespan and increase workflow resource efficiency.

```
# Transform the workflow into a IR DAG
DAG_IR = translate(workflow)
# Apply optimisation on the IR DAG
DAG_IR_OPT = optimise(DAG_IR)
while DAG_IR_OPT has unexecuted operators:
  # Partition the DAG
  partitions = dag_partitioning(DAG_IR_OPT)
  # Get the first partition's operators and the back-end they've
  # been mapped to
  (operators, back_end) = partitions[0]
  # Generate code for the operators
  job_code = generate_code(operators, back_end)
  execute(job_code)
  # Update expected intermediate data sizes
  update_statistics()
  # Remove executed operators from the IR DAG
  remove_operators(DAG_IR_OPT, operators)
```

Listing 3.7: Musketeer scheduling – high-level overview.

To decide which back-ends to use, Musketeer must partition the IR DAG into sub-DAGs, each representing a job. This decision is difficult: on the one hand, the choice of back-ends is affected by the DAG partitioning, and on the other hand, the best partitioning depends on which back-ends are available and what properties these have (e.g., optimised for graph or batch workflows, operator merging abilities). In order for a DAG partitioning solution to be effective, it must:

- take into account back-end expressivity limitations, and partition the DAG only into sub-DAGs that can execute on available back-ends;
- predict with reasonable accuracy the relative performance of available back-ends, and use this prediction to automatically decide which back-ends are "best" for a given workflow; and
- be generalise over limitations of future back-ends in order to not hinder the adoption of new execution engines.

In this section, I explain how my Musketeer prototype meets the above-mentioned requirements. In algorithm 3.7, I give a high-level overview of the steps taken by Musketeer to make decisions. Musketeer starts with the optimised IR operator DAG and iterates as long as there are operators left to execute. In each iteration, Musketeer quantifies the goodness of different DAG partitions using a simple *cost function* that considers information specific to both workflows and back-ends (see §3.5.1). When the input workflow is small or when there are not many operators left to execute, Musketeer quantifies *all* possible DAG partitioning. However, when this is too expensive, Musketeer applies an efficient heuristic based on dynamic programming (§3.5.2). Following, Musketeer generates code for the first sub-DAG (i.e., the job that does not depend on

Parameter	Description
PULL	Rate of data ingest from HDFS.
LOAD	Rate of loading or transforming data.
PROCESS	Rate of processing operator on in-memory data.
PUSH	Rate of writing output to HDFS.

Table 3.2: Rate parameters used by Musketeer's cost function.

other jobs) and executes it in the back-end engine chosen by the partitioning algorithm. When the job completes, Musketeer removes job's operators from the IR DAG and updates statistics (e.g., intermediate data sizes). Musketeer repeats these steps until all operators complete.

3.5.1 Back-end mapping cost function

Musketeer uses a simple *cost function* to compare different DAG partitioning options and to decide which back-end is "best" for each partition. The cost function scores the combination of an operator sub-DAG, input data and back-end system. The cost is finite and represents how long it will take to execute the merged sub-DAG of operators in the given back-end. However, the cost can also be infinite (in practice a large number) if the back-end is not expressive enough to merge the sub-DAG operators. The total cost of a DAG partitioning is the sum of costs of running each sub-DAG partition in its most suitable back-end.

The cost is a weighted sum of four high-level components:

1. **Data volume.** Operators have bounds on their output size based on their behaviour (e.g., whether they is generative or selective). Musketeer applies these bounds to runtime input data sizes to predict intermediate data sizes. Moreover, it refines these predictions after each intermediate workflow job completes and outputs data.

2. **Operator performance.** In a one-off calibration, Musketeer measures each operator in each back-end and records the rate at which it processes input data.

3. **Workflow history.** Many data center workflows are recurrent (e.g., 40% of Bing's workflows [AKB+12b]). Musketeer collects information about each job it runs (e.g., runtime and input/output sizes), and uses this information to refine the scores for subsequent runs of the same workflow.

4. **Back-end resource utilisation efficiency.** If required, Musketeer's cost function can bias the score towards back-ends that may not be fastest, but utilise resources efficiently.

Out of the above-mentioned components, the operator performance provides the most important signal. The operator performance calibration supplies Musketeer with the four rates I list in Table 3.2. PULL and PUSH quantify read and write HDFS throughput. I measure them using

a "no-op" operator. `LOAD`, by contrast, corresponds to back-end-specific data loading or transformation steps (e.g., partitioning the input in PowerGraph). Finally, `PROCESS` approximates the rate at which the operator's computation proceeds. In some systems (e.g., Naiad), I measure `PROCESS` rate directly, while in others (e.g., Hadoop MapReduce), I subtract the estimated duration of the ingest (from `PULL`) and output (from `PUSH`) stages from the overall runtime to obtain `PROCESS`. I use this information to estimate the benefit of shared scans: the cost of `PULL`, `LOAD` and `PUSH` is paid just once (rather than once per-operator), and I combine those with the costs of `PROCESS` for all the operators.

Musketeer conducts the operator performance calibration in a one-off profiling for an idle deployed cluster. Nevertheless, the experiments provide a good indication of how suitable are back-ends for running a particular operator. I do not study how Musketeer operators are affected by co-location interference because I workflows to execute in clusters managed by co-location aware schedulers such as the one one I describe in Chapter 5. These schedulers place workflows in such a way that interference is reduced and workflows complete almost as fast as if they were placed on an idle cluster. In future work, I plan to extend the calibration experiments to consider scenarios in which other co-located jobs are creating contention on resources.

The processing rate parameters Musketeer obtaines from calibration experiments enable generic cost estimates, but Musketeer can make more accurate predictions when it uses workflow-specific historical information (e.g., historical intermediate data size). No such information is available when a workflow first executes, and thus Musketeer only makes conservative performance predictions. For example, when it estimates intermediate operator output data sizes it assumes worst-case (e.g., `SELECT` operators output as much data as they read). Thus, more jobs may initially be generated, but on subsequent workflow executions, Musketeer tightens the bounds using historical information, which may unlock additional merge opportunities and back-end options.

3.5.2 DAG partitioning

There are many ways to divide an IR DAG into partitions (i.e., back-end jobs). The goal of the DAG partitioning algorithm is to find the partitioning with the smallest total cost (i.e., smallest predicted workflow makespan). For each operator, Musketeer must decide if it should or not be merged with the operators that ingest its output. Thus, the number of possibilities to partition a DAG increases exponentially with N, the number of operators in the workflow.

If k, the optimal number of partitions is known a priori, then IR DAG partitioning is an instance of the *k-way graph partitioning* problem [KL70]. However, the optimal number of partitions is unknown a priori, but Musketeer could solver the k-way graph partitioning problem for each k value between one (i.e., all operators execute in a single job) and N (i.e., each operator in a separate job). Unfortunately, the k-way graph partitioning problem is NP-hard [GJS76]: the best partitioning is guaranteed to be found only by exploring all k-way partitions.

Musketeer uses an exhaustive search to find the partitioning with the smallest total cost (§3.5.2.1). In practice, it can only use this search on workflows with up to about 18 operators. For more complex workflows, the exhaustive search is prohibitively expensive and can dominate workflow makespan (see §4.6.1). In such situations, Musketeer switches to a dynamic programming heuristic that completes faster, but does not necessarily finds the optimal partitioning (§3.5.2.2).

3.5.2.1 Exhaustive search

Musketeer explores all the possible DAG partitionings and mappings to back-ends. It takes as input a given IR operator DAG, D, and transforms into a linear ordering of operators. There are many linear orderings for a given DAG, but I choose to topologically sort the operators to obtain an ordering that respects operator precedence – i.e., an operator does not appear in the linear ordering before any of its ancestors. For a given DAG with N operators, the topological sorting algorithm returns a N-tuple of operators $(o_1, o_2, ..., o_n)$.

Following, Musketeer explores all function applications from the linearly ordered N-tuple of operators to $\{0,1\}$ (i.e., $2^{(o_1,o_2,...,o_n)}$) in order to find the best DAG partitioning. It maps each possible operator M-tuple (i.e., $(o_1, o_2, ..., o_m)$), with $M \leq N$, to a corresponding binary representation of numbers from 0 to $2^N - 1$. In this binary representation, an operator is considered to be mapped to a back-end if the binary representation contains a "1" at the operator's position in the linear ordering. For example, a 101 encoding means that the first and third operator are already mapped.

Musketeer uses this representation to efficiently encode the state of mapped operators and to be able to store in a vector C of 2^N length, the minimum cost of each possible state in which zero or more operators are mapped. For each i, $C[i]$ stores the cost of the best mapping for the mapped operators encoded by i. For example, $C[5]$ contains the minimum cost of mapping operator one and three to the best back-end combination. The cost of the best complete DAG partitioning is stored in $C[2^N - 1]$.

Musketeer recursively populates the vector to compute $C[2^N - 1]$. It uses the following recurrence:

$$C[p] = \min_{b \in B \wedge pd, j \in 2^{(o_1,o_2,...,o_n)} \wedge p = pd \oplus j} (C[pd] + cost(j,b))$$

The *cost* function is defined over $2^{(o_1,o_2,...,o_n)} \times B$, where B is the set of supported back-ends. It returns the predicted cost of running a set of operators in a back-end.

Informally, for a given binary mapping p, the algorithm finds the minimum cost partitioning by exploring all the options of partitioning the operators in two partitions: *(i)* a partition pd for which the algorithm previously computed the minimum cost, and *(ii)* a partition j in which all the operators are mapped to a job that would execute in back-end b. The algorithm is guaranteed

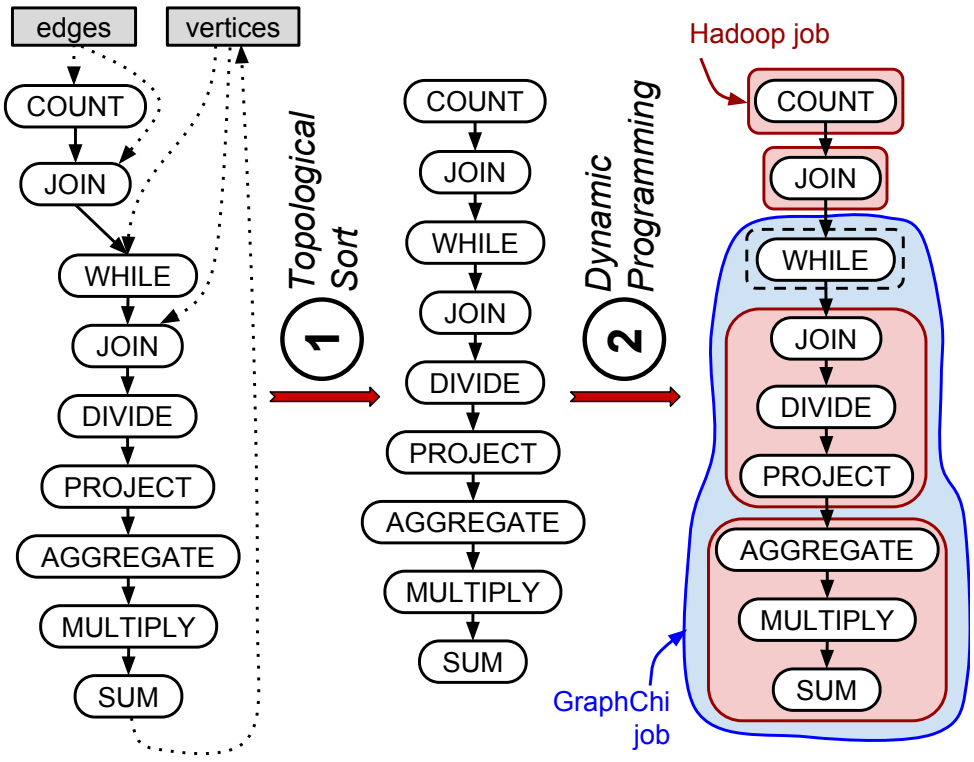


Figure 3.7: The dynamic partitioning heuristic takes an IR DAG, *(1)* transforms it to a linear order, and *(2)* computes job boundaries via dynamic programming. On the right, I show several possible partitions and system mappings.

to find the optimal partitioning with respect to the cost function because it explores all the possible partitions and mappings.

The algorithm is exponential in the number of workflow operators (N) because it computes the minimum cost mapping for each possible binary encoding between 0 and $2^N - 1$. Its worst-case complexity is $O(2^N * |B|)$. Thus, Musketeer can only use the algorithm on workflows with up to few dozens operators because it would otherwise significantly increase workflow makespan.

3.5.2.2 Dynamic programming heuristic

Some industry workflows comprise of large DAGs that comprise of up to hundreds of operators [CRP+10, §6.2]. Musketeer partitions such workflows with a dynamic programming heuristic I developed. The algorithm chooses good partitionings and mappings in practice, and its execution time scales linearly with the number of operators. It focuses on grouping consecutive operators within the linear ordering of operators into jobs, and thus explores only a subset of the possible partitionings.

In Figure 3.7, I illustrate the high-level steps of my algorithm on the IR operator DAG of the PageRank workflow. First, the algorithm linearly orders the IR operator DAG. Second, it explores possibilities of merging neighbouring operators into single jobs using a dynamic programming algorithm that finds the optimal partitioning of the ordering.

Not all linear operator orderings are equally good inputs to the dynamic programming algorithm. In order for the algorithm to find good partitionings, the linear ordering must have the following properties: *(i)* it must respect operator precedence (i.e., an operator does not appear in the linear ordering before any of its ancestors), and *(ii)* it must place in neighbouring positions as many mergeable operators as possible. Topological sorting algorithms produce orderings that have the above-mentioned properties. I implemented a topological sort algorithm that is based on depth-first search because Musketeer runs more workflows that have greater IR DAG "height" rather than "width". The algorithm produces orderings that provide more operator merging opportunities than other alternatives.

My dynamic programming algorithm uses a cost function (see § 3.5.1) to decide which back-end is "best". For a given n-tuple of operators, $(o_1,\ o_2,\ \ldots,\ o_n)$, $cost((o_1,\ o_2,\ \ldots,\ o_n), b)$, estimates how long it would take until the operators complete if they were to run as a single job on back-end b. The algorithm uses this function to compute for each N-operator workflow a N-by-N matrix C. For each $0 < n_{jobs} \leq n_{ops} \leq N$, the matrix stores at $C[n_{ops}][n_{jobs}]$ the minimum cost of running the first n_{ops} linearly ordered operators using n_{jobs} jobs on the "best" back-ends. Musketeer computes the matrix with the following recurrence:

$$C[n_{ops}][n_{jobs}] = \min_{b \in B \wedge op_{exec} < n_{ops}} \left(C[op_{exec}][n_{jobs} - 1] + cost((o_{op_{exec}+1},\ \ldots,\ o_{n_{ops}}), b) \right)$$

Informally, the dynamic programming algorithm determines the best partitioning for running the first n_{ops} operators using n_{jobs}. It explores all the possibilities for running the first op_{exec} operators of the ordering using $n_{jobs} - 1$ and the remainder operators as a single job. The algorithm finds good solutions because it considers all partitionings of the linear orderings and because the cost function ensures it merges as many operators as possible within each individual job.

In Listing 3.8, I show the pseudocode of my implementation. My algorithm computes two auxiliary N-by-N matrices BE and $JOBS$ besides the matrix C. The algorithm uses these matrices to reconstruct the best partitioning and back-end mappings upon its completion. For each $0 < n_{jobs} \leq n_{ops} \leq N$, $BE[n_{ops}][n_{jobs}]$ stores the name of the back-end it chose for last job, and $JOBS[n_{ops}][n_{jobs}]$ stores the binary encoding of the last job. In contrast to the exhaustive search, the dynamic programming heuristic scales to large workflows because it has a polynomial worst-case complexity of $O(N^3 * |B|)$. However, the heuristic can miss out on opportunities to merge operators because the linear orderings it chooses may not contain the best operator adjacencies. I discuss this further and show an example in §3.6. Nonetheless, I found the dynamic programming heuristic to work well in a set of cluster experiments I conducted (§4.6.2).

```
for n_ops in range(1, N + 1):
  for n_jobs in range(1, n_ops + 1):
    # Vary the number of operators included in the last job
    for job_size in range(1, n_ops - n_jobs + 1):
      # Tries mapping the last job in each back-end
      for back_end in back_ends:
        # Compute the binary encoding of the last job
        job_encoding = (1 << job_size - 1) << n_ops - job_size
        # Update the cost if it is smaller than the cost of the
        # best partitioning for the first n operators
        if (C[n_ops - job_size][n_jobs - 1] + cost(job_encoding, back_end) <
            C[n_ops][n_jobs]):
          C[n_ops][n_jobs] = C[n_ops - job_size][n_jobs - 1] +
                             cost(job_encoding, back_end)
          # Store the back-end to which the last job has been mapped
          BE[n_ops][n_jobs] = back_end
          # Store last job's binary encoding
          JOBS[n_ops][n_jobs] = job_encoding
```

Listing 3.8: Dynamic programming heuristic for exploring partitionings of large workflows.

3.6 Limitations and future work

Decoupling front-end frameworks from back-end execution engines increases flexibility, but it may obfuscate some end-to-end optimisation opportunities that expert users can write. Musketeer is best suited for non-specialist users that write analytics workflows for high-level front-end frameworks. Users who are willing to painstakingly hand-optimise a particular workflow are not Musketeer's target audience.

In the following paragraphs, I highlight some of Musketeer's current limitations and discuss how they can be addressed in future work.

Expressing workflows (§3.2). Musketeer currently expects users to define graph processing workflows in either Lindi, BEER, or its GAS DSL. However, many vertex-centric graph processing systems do not constrain users to express workflows using only sequences of relational operators, but allow them to provide arbitrary per-vertex code implemented in high-level programming languages (e.g., Java for Giraph, C++ for GraphChi). Musketeer uses user-defined functions (UDFs) to run such workflows. This limits Musketeer's choice of back-ends that can execute the user-provided code, and it may cause Musketeer to generate code that uses inefficient foreign-function interfaces. In the future, I plan to improve Musketeer's support for user-defined functions. I could use source-to-source compilers to translate UDFs to each back-end's programming language of choice, or I could automatically translate UDFs into relational operators using constraint-based synthesis [IZ10; CSM13].

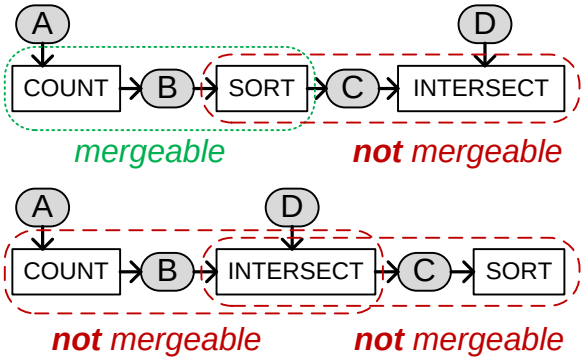


Figure 3.8: Example of a DAG optimisation inadvertently decreasing operator merging opportunities.

Optimising workflows (§3.3.1). The optimisations I described in Subsection 3.3.1 re-order operators with the goal of reducing intermediate data sizes. However, these optimisations are not always beneficial: they can reduce operator merging opportunities or change the IR DAG such that Musketeer's dynamic programming heuristic does not find good partitionings.

Consider the example of the workflow shown in Figure 3.8: it is composed of three operators: `COUNT`, `SORT` and `INTERSECT`. Musketeer's query rewriting rules would pull the selective `INTERSECT` operator above `SORT` operator. But this may be unhelpful – while `COUNT` and `SORT` can be merged in MapReduce back-ends, `COUNT` and `INTERSECT` cannot. Applying this rewrite would have the side-effect of eliminating the possibility of merging the two operators in any MapReduce-based back-end. The workflow would incur extra job startup costs and would necessitate three passes over the data rather than two.

To avoid this problem, Musketeer's query optimiser would have to return the entire set of reordered IR operator DAGs. Musketeer could then apply the dynamic programming heuristic algorithm on each DAG and pick the partitioning with the smallest cost. If the number of obtained DAGs by applying the query rewriting rules is too large, Musketeer could only partition the most promising DAGs.

Idiom recognition (§3.4.2). Some graph workflows can be expressed as an iterative computation or as a unrolled sequence of relational operators. For example, a workflow counting edge triangles in a graph can be expressed as: *(i)* an iterative graph computation, or *(ii)* as a computation that joins the edges twice, filters out the triangles and counts them. In the latter case, Musketeer fails to recognise that triangle counting is a graph workflow and does not run it on a specialised graph processing back-end. This limitation could partly be solved with a "reverse loop unrolling" heuristic that detects when multiple operators take the same input and produce the same output (or a closure thereof).

Back-end mapping cost function (§3.5.1). The cost function mainly uses signals that are statically computed before runtime (e.g., operator processing parameters computed in one-off

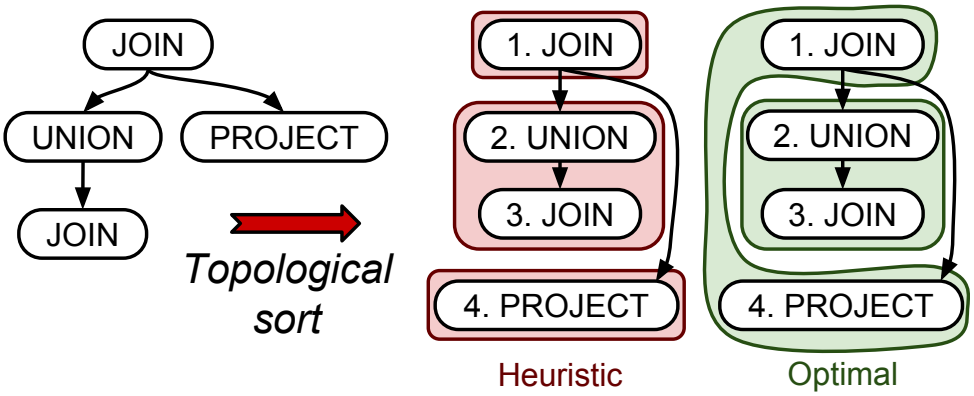


Figure 3.9: The dynamic heuristic does not return the minimum-cost partitioning for this workflow: it misses the opportunity to merge `JOIN` with `PROJECT`.

calibration experiments), or are only updated upon job completion (e.g., intermediate data sizes, historical performance). The cost function could better predict makespans if it were to consider resource utilisation at the nodes on which the back-ends execute. Similarly, the cost function could reduce workflow makespan if it were to monitor back-ends' state and take into account current job queue length when scoring sub-DAGs. It is common for back-ends to have long queues of jobs waiting to execute [ZBS+10]. In such cases, choosing a suboptimal back-end may reduce workflow makespan.

DAG partitioning (§3.5.2). My dynamic programming heuristic returns the optimal k-way partitioning for a given linear ordering of operators. However, it may miss fruitful merging opportunities because it only explores one linear ordering. In Figure 3.9, I show a workflow example on which the dynamic heuristic misses on good merging opportunities. In Hadoop MapReduce, it is best to run the top `JOIN` in the same job as the `PROJECT`, but the linear ordering based on depth-first exploration breaks this merge opportunity. This limitation does not affect general-purpose back-ends (e.g., Naiad and Spark), which are able to merge any sub-region of operators. A simple approach to mitigate this limitation is to generate multiple linear orderings, run my inexpensive dynamic partitioning heuristic for each one of them, and pick the best partitioning.

3.7 Summary

In this chapter, I argue that data processing systems should move to an architecture in which front-end frameworks are decoupled from back-end execution engines. I first give an overview of Musketeer (§3.1) and then I describe how it decouples systems by translating workflows defined in one of the supported front-end frameworks (§3.2) to an intermediate representation consisting of a dynamic DAG of operators (§3.3). Following, I present how Musketeer applies optimisations and generates code for suitable back-end execution engines (§3.4). Next, I de-

scribe how Musketeer automatically partitions the IR DAG and decides which combination of back-ends is "best" for running a workflow (§3.5). Finally, I highlight Musketeer's limitations and discuss several ways in which these could be addressed (§3.6)).

With Musketeer, users benefit from increased flexibility: workflows can be written once and mapped to many back-ends, different back-ends can be combined within a workflow and existing workflows seamlessly ported to new execution engines. In Chapter 4, I show that Musketeer enables compelling performance gains and its generated code performs almost as well as unportable, hand-optimised baseline implementations.

Chapter 4

Evaluation Musketeer

Musketeer's goals are to reduce workflow makespan and resource utilisation. In order to achieve them, Musketeer: *(i)* maps workflows expressed in front-end frameworks to its intermediate representation, *(ii)* dynamically and automatically chooses combinations of back-ends to run workflows on, and *(iii)* generates efficient back-end code.

In this chapter, in a range of cluster experiments with real-world workflows, I show that Musketeer meets its goals. In my experiments, I answer the following six questions:

1. How does Musketeer's generated workflow code compare to hand-written optimised implementations? (§4.2)
2. Do Musketeer's operator merging and look-ahead optimisations reduce workflow makespan? (§4.3)
3. Does Musketeer manage to speedup legacy workflows by mapping them to a different back-end execution engine? (§4.4)
4. Does Musketeer reduce workflow makespan by flexibly combining back-end execution engines? (§4.5)
5. How fast is Musketeer's automatic execution engine mapping? (§4.6.1)
6. Does Musketeer's automatic execution engine mapping make good choices? (§4.6.2)

In my experiments, I use seven real-world workflows: three batch workflows, three iterative workflows and a hybrid batch-graph workflow. The batch workflows run: *(i)* TPC-H query 17, *(ii) top-shopper*, which identifies an online shop's top spenders, and *(iii)* a Netflix movie recommendation algorithm. The iterative workflows run: *(i)* PageRank, *(ii)* single-source shortest path (SSSP), and *(iii) k*-means clustering; Finally, the hybrid workflow comprises of a batch-preprocessing step followed by PageRank.

4.1 Experimental setup and metrics

I execute all the experiments I describe in this chapter on either a heterogeneous local cluster or on a medium-sized Amazon Elastic Compute (EC2) cluster. The clusters differ both in hardware and scale:

The heterogeneous local cluster is a small seven-machine cluster comprising of machines with different CPU clock frequencies and from different generations. The machines also have different memory architectures and RAM capacities. All the machines are connected via a two-switch 1G network (see Table 4.1a).

The medium-sized cloud cluster is a cloud computing cluster consisting of 100 m1.xlarge Amazon EC2 instances (see Table 4.1b). The instances have 15GB of RAM and four virtual CPUs. The cluster represents a realistic environment in which the network and the machines are shared with other EC2 tenants.

The local cluster models a small, albeit realistic heterogeneous cluster in which machines have different architectures and resource capacities (see §2.3.1.1). All the experiments I conduct on the local cluster are executed in a fully controlled environment without any external network traffic or machine utilisation. I use this environment to measure with low variance frameworks' performance and their suitability to execute workflows on heterogeneous clusters. In contrast, the medium-sized cloud cluster represents a realistic data center environment in which workflows can be affected by interfering applications running in other tenants' instances. In both clusters, all machines run Ubuntu 14.04 (Trusty Tahr) with Linux kernel v3.14.

I deploy all systems supported by Musketeer[1] on these clusters. I use a shared Hadoop File System (HDFS) as the storage layer because the file system is already supported by Hadoop, Spark and PowerGraph. Moreover, in order to establish a level playing field for my experiments, I tune, modifiy, and add support for interaction with HDFS in all the other frameworks (see Table 4.2).

4.1.1 Metrics

In my experiments, I focus on three workflow metrics that I now describe.

Makespan is an end-to-end metric that measures the total execution time of workflows (i.e., from when a workflow is submitted until its output is completly written in HDFS). Makespan includes the time it takes to load input data from HDFS, to pre-process or transform data to the formats supported by back-ends (e.g. graph partitioning and sorting in PowerGraph and GraphChi), workflow computation time, and the time it takes to write the output to HDFS. As

[1] Hadoop 2.0.0-mr1-chd4.5.0, Spark 0.9, PowerGraph 2.2, GraphChi 0.2, Naiad 0.2 and Metis commit `e5b04e2`.

	Type	Machine	Architecture	Cores	Thr.	Clock	RAM
3×	**A**	GW GR380	Intel Xeon E5520	4	8	2.26 GHz	12 GB PC3-8500
2×	**C**	Dell R420	Intel Xeon E5-2420	12	24	1.9 GHz	64 GB PC3-10666
1×	**D**	Dell R415	AMD Opteron 4234	12	12	3.1 GHz	64 GB PC3-12800
1×	**E**	SM AS1042	AMD Opteron 6168	48	48	1.9 GHz	64 GB PC3-10666

(a) Heterogeneous seven-machine local cluster.

	Instance type	Compute Units (ECU)	vCPUs	RAM
100×	m1.xlarge	8	4	15 GB

(b) Medium-sized, multi-tenant Amazon EC2 cluster.

Table 4.1: Specifications of the machines in the two evaluation clusters.

System	Modification
Hadoop	Tuned configuration to best practices.
Spark	Tuned configuration to best practices.
GraphChi	Added HDFS connector for I/O.
Naiad	Added support for parallel I/O and HDFS.

Table 4.2: Modifications made to back-end execution engines deployed.

a result, the numbers I present are not directly comparable to those presented in many papers, which show just the workflow computation time. I believe that makespan is a more insightful metric to use than computation time because it includes all the overheads caused by pre and post-processing. Makespan is an accurate estimation of how long users would have to wait for their workflows to complete.

Resource efficiency is a measure of the efficiency loss incurred due to scaling out over multiple machines and executing workflows in unoptimised frameworks. I compute resource efficiency by normalising workflows' fastest single-machine makespan (I assume it to be maximally resource-efficient) to their aggregate execution time over all machines when running in a distributed back-end. For example, a workflow that runs for 30s using all 100 nodes of the EC2 cluster has an aggregate execution time of 3,000s. If the best single-node back-end completes the same workflow in 2,000s, the resource efficiency of the distributed execution is 66%.

Finally, I also measure the workflow **overhead** introduced by Musketeer generated code. I calculate overhead by normalising the makespan of Musketeer generated workflows executing in a single framework to an optimised hand-written implementation for the same framework.

4.2 Overhead over hand-written, optimised workflows

For Musketeer to be attractive to developers, the code it generates must add minimal overhead over an optimised hand-written implementation. Otherwise, Musketeer code overheads may outweigh makespan differences arising from back-end execution engines' design choices. It is

not difficult to generate code that is competitive with hand-written implementations for single-operator workflows, but it is challenging to do it for multi-operator workflows.

Musketeer combines techniques such as operator merging (§ 3.4.1), idiom recognition (§ 3.4.2), data scans sharing (§ 3.4.3) and type inference (§ 3.4.4) in order to optimise the code it generates for multi-operator workflows. In the following, I show using a complex batch workflow and an iterative workflow that Musketeer's generated code overhead over optimised hand-written baselines does not exceed 30% and is usually around 5–20%.

4.2.1 Batch processing

I first measure Musketeer's generated code overheads using the batch Netflix movie recommendation workflow [BKV08]. The workflow implements collaborative filtering for movie recommendation (see Figure 4.1). It takes two inputs: a 100 million-row movie ratings table (2.5GB) and a 17,000-row movie list (0.5MB). From these inputs, the collaborative filtering workflow computes movie recommendations for all users, and finally outputs the top recommended movie for each user. It is a challenging workflow because it contains many operators (13) and provides several, but non trivial, operator merging opportunities. Moreover, the workflow is data-intensive, with up to 700 GB of intermediate data generated. These data are large enough to not be possible to quickly process it on a single machine, but small enough to fit within my medium-sized EC2 cluster's memory.

In my experiment, I look at how Musketeer's generated code compares to hand-written baselines when it processes different amounts of data. I control the amount of processed data by varying the number of movies I use for computing recommendations. For example, the workflow generates only 17 GB of intermediate data when it uses approximately 700 movies for computing recommendations. Whereas, the intermediate data size increases to 240 GB when the algorithm uses 1,200 movies, and to 700 GB when it uses 3,500 movies.

I hand-implemented workflow baselines in three general-purpose systems that are good candidates for running the workflow (Hadoop MapReduce, Spark and Lindi on Naiad). Hadoop MapReduce is a good candidate because it can quickly process large amounts of data, but to its disadvantage, it has to write intermediate data to disks several times because it cannot run the entire workflow in a single job. By contrast, Spark only spills intermediate data to disk if the data does not fit in memory. Finally, Lindi on Naiad is a good candidate because it can execute the entire workflow in a single job and it can stream input and intermediate operators' output data to depedent operators as soon as the data are available.

In Figure 4.2, I compare Musketeer-generated code for the Netflix workflow to the aforementioned hand-optimised baselines. I extensively tune each baseline to deliver good performance for the given system. I use system-specific optimisations and take advantage of the above-mentioned system properties.

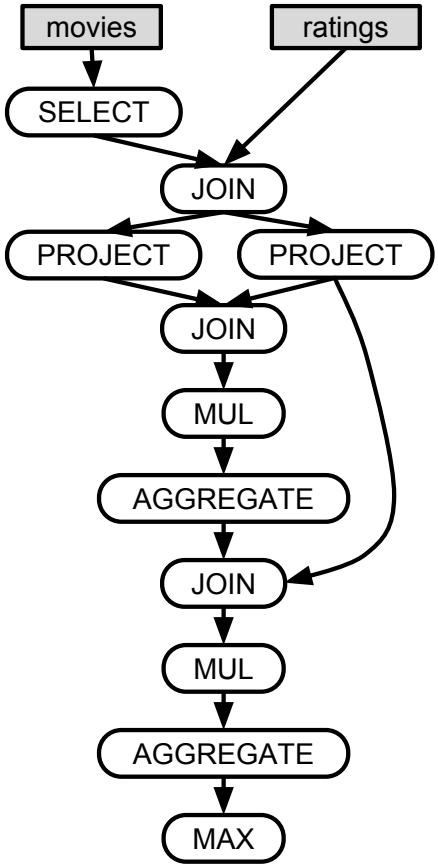


Figure 4.1: Netflix movie recommendation workflow.

For all three systems, the overhead added by Musketeer's generated code is low: it is almost non-existent for Naiad and is under 30% for Spark and Hadoop even as the input and intermediate data grow. The remaining overhead for Spark is primarily due to my type-inference algorithm's simplicity, which can cause the Musketeer-generated code to make an extra pass over the data. However, I plan to reduce this overhead in future work.

4.2.2 Iterative processing

I now measure Musketeer's generated code overheads on iterative workflows using the PageRank algorithm. Iterative workflows are challenging because Musketeer must generate code that efficiently uses each back-end execution engine's mechanism for running iterative computations. For example, Musketeer must generate code for Spark's driver program, generate code that uses Naiad's timely dataflow model, or run its own driver program to execute iterative computations on Hadoop MapReduce.

In my experiment, I look at how Musketeer's generated code compares to hand-written optimised baselines. I hand-implemented baselines for the most popular distributed and single-machine back-end execution engines supported by Musketeer that can run the PageRank workflow: Hadoop MapReduce, Spark, Naiad, PowerGraph, GraphChi. The workflow executes 5

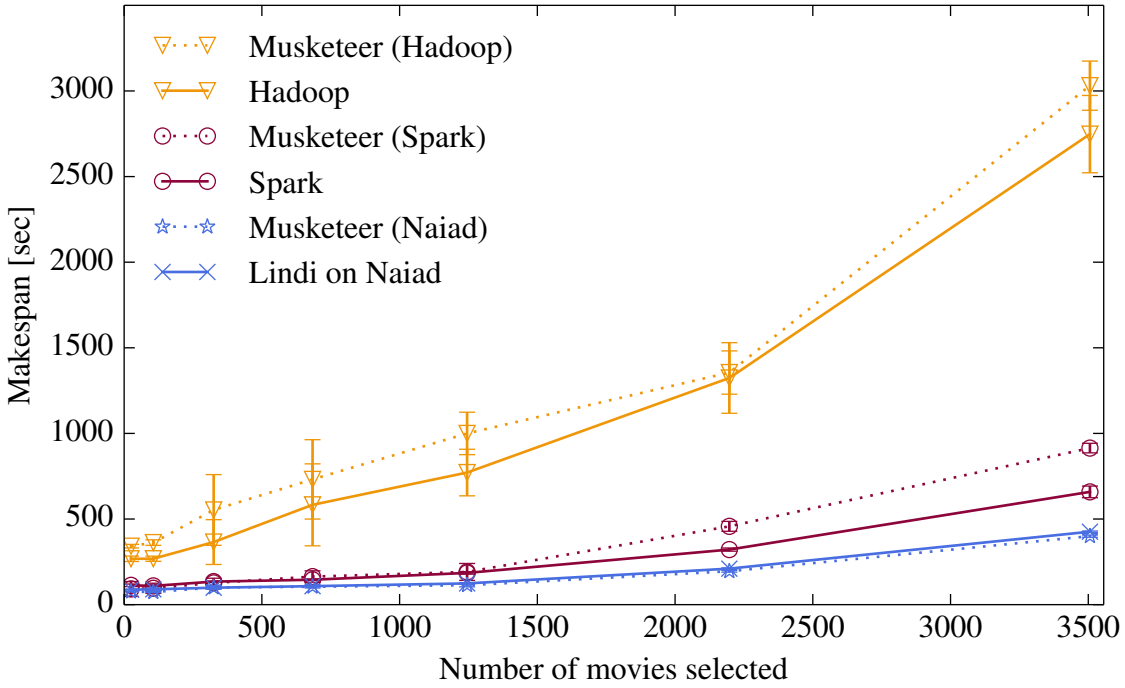


Figure 4.2: Makespan of Musketeer-generated code versus hand-written implementations for back-ends on the Netflix movie recommendation workflow. Error bars show $\pm\sigma$ over five runs on the 100 EC2 instances cluster.

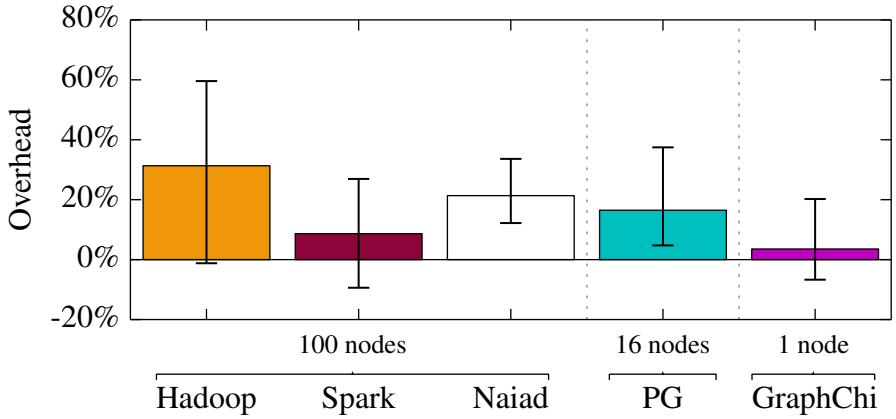


Figure 4.3: Musketeer-generated code overhead for PageRank on the Twitter graph. I use three cluster setups: a medium-sized 100-node EC2 cluster, 16-node EC2 cluster on which PowerGraph runs best, and a single EC2 node for GraphChi. Error bars show $\pm\sigma$ over five runs; negative overheads are due to variance on EC2.

PageRank iterations on the Twitter graph with 43 million vertices and 1.4 billion edges (23 GB of input data). In Figure 4.3, I show the overheads of Musketeer-generated code over the baselines. The average overhead is below 30% in all cases. The variability in overhead (and improvements over the baseline) are due to interference with other users' instances and to performance variance on EC2.

There are no fundamental reasons for Musketeer to generate code even with overhead as little as 30%. Indeed, further optimisations of the code generation are possible. For example,

Musketeer-generated code currently does several unnecessary string operations per input data row even when operators are merged. These operations could be implemented more efficiently or removed in some cases. Most such optimisations would benefit *all* code Musketeer generates for a particular back-end.

In conclusion, my experiment shows that Musketeer generates code that performs nearly as well as hand-written optimised baselines for both batch and iterative processing workflows. This together with the ability to dynamically explore multiple execution engines, and with the improved portability Musketeer offers, make for a compelling case.

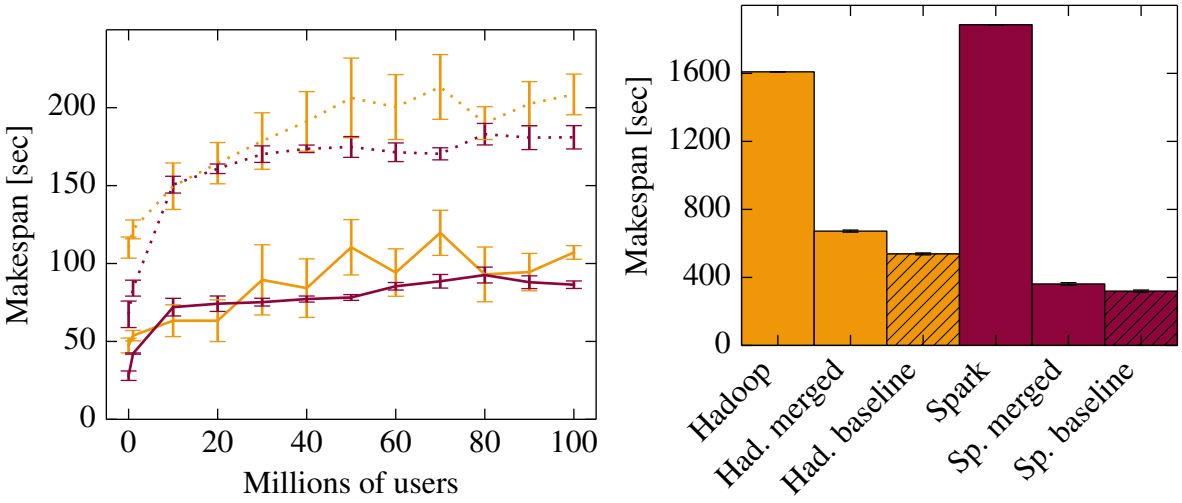
4.3 Impact of Musketeer optimisations on makespan

Musketeer takes workflow specifications, translates them into its IR, generates code, and executes the code in one or more of the back-ends it supports. A simple way to execute workflows is to generate code and run a back-end job for each operator. But this approach greatly increases workflow makespan.

Instead, Musketeer uses three key techniques to improve the code it generates: *(i)* operator merging (§ 3.4.1), *(ii)* data scans sharing (§ 3.4.3), and *(iii)* look-ahead and type inference (§ 3.4.4). With operator merging, Musketeer executes several operators in a single job, and thus avoids job creation overheads where possible. Moreover, by sharing data scans Musketeer avoids superfluous reads and writes of intermediate data. Finally, by using look-ahead and type inference Musketeer is able to transform each operator's output to match input requirements of depedent merged operators. This technique reduces the amount of string processing workflows conduct. I measure the effect these optimisations have on workflow makespan using a simple micro-benchmark *top-shopper* batch workflow and a complex iterative workflow.

```
1  SELECT purchase_id, amount, user_id
2    FROM shop_logs
3    WHERE country = 'USA'
4    AS usa_shop_logs;
5  SELECT user_id, SUM(amount) AS total_amount
6    FROM usa_shop_logs
7    GROUP BY user_id
8    AS usa_spenders;
9  SELECT user_id, total_amount
10   FROM usa_shoppers
11   WHERE total_amount > 12000
12   AS usa_big_spenders;
```

Listing 4.1: Hive code for *top-shopper* workflow.



(a) *top-shopper* workflow running on the 100-node EC2 cluster.

(b) Cross-community PageRank on the local heterogeneous cluster.

Figure 4.4: Operator merging and type inference eliminate per-operator job creation overheads and help bring generated code performance close to hand-written baselines.

4.3.1 Batch processing

In Listing 4.1, I include the Hive query code for the micro-benchmark *top-shopper* workflow. The workflow finds the largest spenders in a certain geographic region. It first filters a set of purchases by region, then it aggregates their values by user ID. Finally, it selects all users that have spent over a threshold.

Top-shopper consists of three `SELECT` operators. The first and third operators have only simple `WHERE` clauses, whereas the second operator has a `GROUP BY` clause. Given that the workflow contains only one `GROUP BY` clause, all its operators can be merged and executed as a single job even in the least expressive supported back-end (i.e., Hadoop MapReduce).

In Figure 4.4a, I show top-shopper's makespan with operator merging, data scans sharing and type inference turned off and on. I also vary the size of the input data from few KB up to 30 GB (100 million users) to show how much these optimisations help as I increase input data size. The results illustrate that the above-mentioned optimisations significantly reduce makespan: I observe a one-off reduction in makespan of ≈25–50s due to avoiding per-operator job creation overheads and sharing data scans, along with an additional 5–10% linear benefit per 10M users attributable to type inference which reduces the number of string operations conducted.

4.3.2 Iterative processing

I now measure by how much the three optimisations reduce overheads using a complex *cross-community PageRank* workflow. The workflow computes the relative popularity of users present in two web communities. It involves a batch computation followed by an interactive computation. First, the workflow intersects the edge sets of two communities (e.g., all LiveJournal and

WordPress users), and subsequently, it runs five PageRank iterations on the edges present in both communities. In contrast to the top-shopper workflow, cross-community PageRank cannot execute as a single job in the least expressive back-end (Hadoop MapReduce). Thus, the workflow is a challenging workflow, because Musketeer must automatically decide which operators can merged together and what is the best way to merge them.

In my experiment, I use as input the LiveJournal graph (4.8M nodes and 68M edges) and a synthetically generated web community graph (5.8M nodes and 82M edges). In Figure 4.4b, I show that both Hadoop MapReduce and Spark generated code for cross-community PageRank see a benefit. Each Hadoop MapReduce iterations completes 180 seconds faster and each Spark iteration completes 300 seconds faster because of operator merging, data scans sharing and type inference. To sum up, these three optimisations reduce workflow makespan by up to 60% for the simple top-shopper batch workflow and up to 80% for the complex iterative cross-community PageRank workflow.

4.4 Dynamic mapping to back-end execution engines

I have previously shown that no data processing framework systematically outperforms all other at different scales (see § 2.2.3). I now investigate Musketeer's ability to dynamically map workflows to the most competitive back-end execution engine at different cluster scales. I consider a batch workflow and an iterative graph workflow to show that Musketeer can leverage back-end diversity for different types of workflows.

4.4.1 Batch processing

I run query 17 from the TPC-H business decision benchmark using the HiveQL and Lindi front-ends to illustrate the flexibility offered by Musketeer at different scales. The TPC-H benchmark generates data to accurately model the day-to-day operations of an online wholesaler. Query 17 is a business intelligence query that computes how much yearly revenue would be lost if the wholesaler would not accept small orders. It first selects all parts of a given brand and with a given container type. Following, it computes the average yearly order size for these parts. Finally, the query calculates the average yearly loss in revenue if orders smaller than 20% of this average were no longer taken. I implemented the query in HiveQL using one `SELECT` with a `GROUP BY` clause, two `JOIN`s and a `SELECT` with a simple `WHERE` clause. I use equivalent LINQ operators to implement the workflow in Lindi.

In Figure 4.5, I show workflow makespan as I increase input data size from 7.5 GB (TPC-H scale factor 10) to 75 GB (TPC-H scale factor 100). Makespan ranges between 200–400s when I run the Hive version of the workflow using Hive's native Hadoop MapReduce back-end. Musketeer, however, can map the Hive workflow specification to different back-ends. In this case,

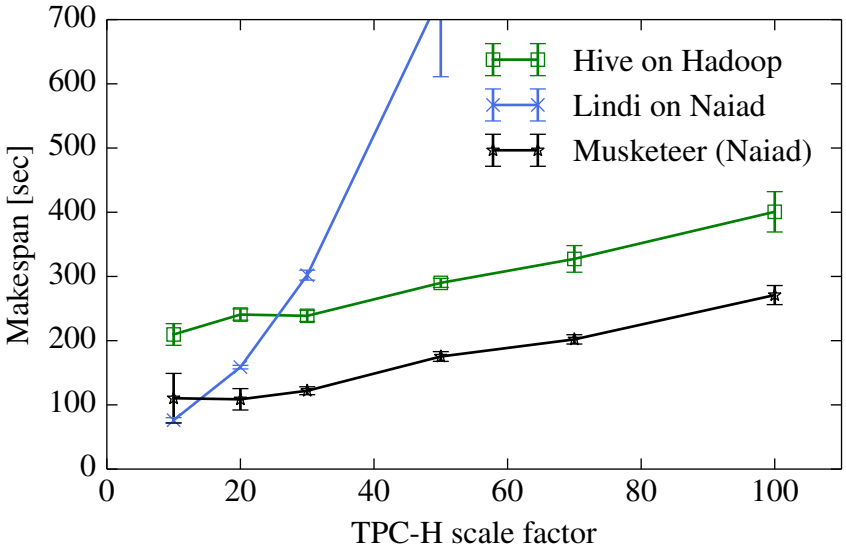


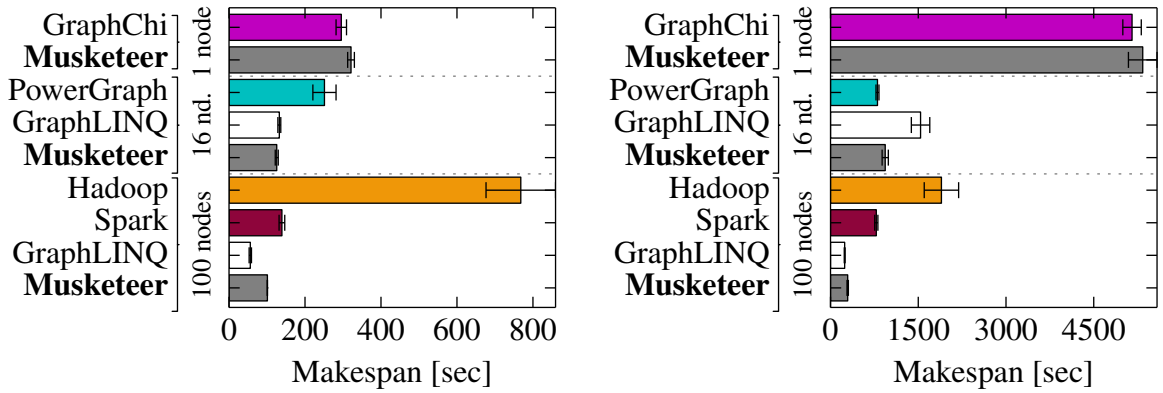
Figure 4.5: Musketeer reduces TPC-H query 17 makespan on a 100-node EC2 cluster compared to Hive and Lindi running jobs on their native back-ends. Less is better; error bars show min/max of three runs.

if Musketeer maps it to Naiad then it reduces the makespan by 2×. This is not surprising: Hive cannot run the workflow with fewer than three Hadoop MapReduce jobs because MapReduce is based on the restrictive bipartite dataflow model, while Naiad can run the entire workflow in a single job.

A developer might have sufficient understanding of data processing systems to know that Naiad can run the workflow in a single job. He could specify the workflow using the Lindi front-end and target Naiad directly. However, the Lindi query scales worse than the Hive query, despite the query running as a single job on Naiad. This is the case because Lindi's high-level `GROUP BY` operator is non-associative. Thus, the Naiad job generated for the workflow's Lindi version collects all of `GROUP BY`'s input data on a single machine before it applies the operator. This limitation does not affect Musketeer which generates code for an improved associative `GROUP BY` operator implemented using Naiad's low-level vertex API. The associative operator groups data locally on each machine before it sends the data to one machine. Consequently, Musketeer's generated Naiad code scales far better than the Lindi version (up to 9× at scale 100). The Naiad developers may of course improve Lindi's `GROUP BY` in the future, but this example illustrates that by decoupling and generating efficient code, Musketeer can improve performance even for a front-end's native execution engine.

4.4.2 Iterative processing

While batch workflows can be expressed using SQL-like front-end frameworks such as Hive and Lindi, iterative graph processing workflows are typically expressed in other types of front-ends (see §3.2.3). To evaluate Musketeer's ability to dynamically map graph computations at different scales, I implemented PageRank using Musketeer's GAS DSL front-end (Listing 3.4,



(a) Orkut (3.0M vertices, 117M edges).

(b) Twitter (43M vertices, 1.4B edges).

Figure 4.6: Musketeer performs close to the best-in-class system for five iterations of PageRank on 1, 16 and 100 EC2 nodes. Error bars are $\pm\sigma$ over 5 runs.

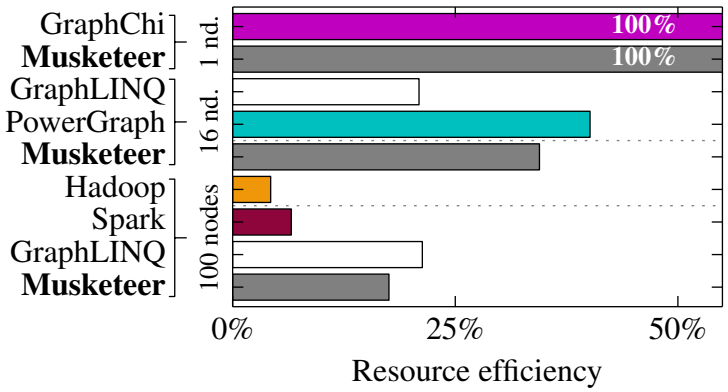


Figure 4.7: Musketeer's resource efficiency on PageRank on the Twitter graph (more is better).

§3.2.3). I run this workflow on the two social network graphs (Orkut and Twitter) I priorly compared the data processing systems (see §2.2.3).

In Figure 4.6, I compare the makespen of five PageRank iterations executed using Musketeer-generated jobs to hand-written optimised baselines implemented in: *(i)* general-purpose systems (Hadoop, Spark), *(ii)* a specialised graph-processing front-end for Naiad (GraphLINQ), and *(iii)* special-purpose graph processing execution engines (PowerGraph, GraphChi). Different systems achieve their best performance at different scales, and I only show the best result for each system. The only exception to this is GraphLINQ on Naiad, which is competitive at both 16 and 100 nodes. At each scale, Musketeer's best mapping is almost as good as the best-in-class baseline. On one node, Musketeer does best when mapping to GraphChi, while mapping to Naiad (Orkut) or PowerGraph (Twitter) is best at 16 nodes, and mapping to Naiad is always best at 100 nodes.

4.4.2.1 Resource efficiency

Workflow makespan is an important metric for many users, however, in oversubscribed clusters it may be worthwhile to trade-off makespan for improved resource efficiency. In Figure 4.7, I show the resource efficiency for the same configurations for running PageRank on the Twitter graph. Musketeer achieves resource efficiencies close to the best stand-alone implementations at all three scales. On one node Musketeer generates GraphChi code that is as efficient as the baseline. On sixteen machines, Musketeer is 14% less resource efficient than the most efficient baseline executed in PowerGraph. Finally, on 100 machines, Musketeer is 17% less resource efficient than the best alternative (GraphLINQ).

In conclusion, these experiments demonstrate that Musketeer's dynamic mapping approach is flexible and can be used for both batch and iterative computations. Musketeer can either reduce makespan of legacy workflows or execute them in more efficient back-ends.

4.5 Combining back-end execution engines

In addition to dynamically mapping *entire* workflows to specialised back-ends, Musketeer can also combine different back-ends by mapping *parts* of complex workflows to different back-ends. Musketeer can leverage execution engine diversity and use systems only for the parts of workflows they have been specialised for. Prior workflow managers (e.g., Oozie, Pig) can only execute complex workflows that require both batch and iterative graph processing in a single back-end (e.g., Hadoop MapReduce), whereas if worthwhile, Musketeer can execute the workflow's batch computation in a general data processing back-end and its graph computation in a specialised graph processing back-end.

I investigate Musketeer's capability to combine different back-end execution engines using the *cross-community PageRank* workflow. This workflow yields the relative popularity of the users present in both of two web communities. It comprises of a batch operator (`INTERSECTION`) that finds the users and links present in two communities, and an implementation of the PageRank algorithm that computes these users' popularity. In my experiment, I use the LiveJournal Graph (4.8M nodes and 68M edges) and a synthetically generated web community graph (5.8M nodes and 82M edges). I run five PageRank iterations over the intersection of these two graphs on the local heterogeneous cluster using Musketeer-generated code for different back-ends. All back-ends run on all seven machines, with the exception of PowerGraph which runs on two machines (best configuration) and GraphChi which is a single-machine back-end.

In Figure 4.8, I compare makespan of cross-community PageRank workflow for different combinations of back-ends, explored using Musketeer. [2]

Out of the three executions on single back-ends, the workflow completes fastest in Lindi in 153s. However, the makespan is comparable when Musketeer combines Hadoop MapReduce with a

[2]The 100-instances EC2 cluster had similar results, albeit at increased variance.

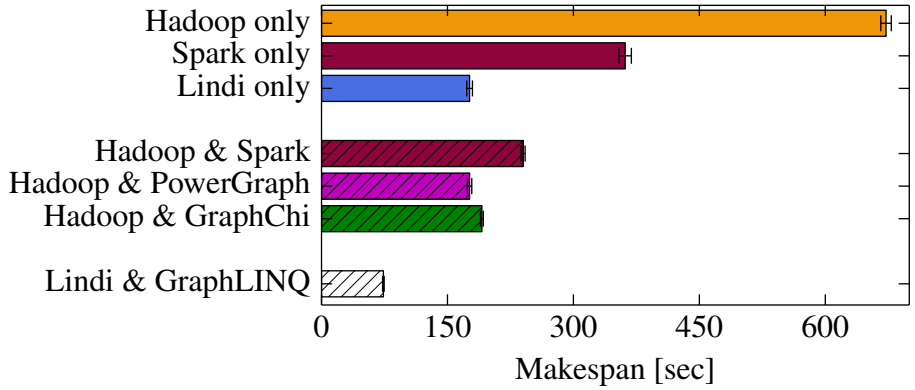


Figure 4.8: A cross-community PageRank workflow is accelerated by combined back-ends. All jobs apart from the "Lindi & GraphLINQ" combination were generated by Musketeer. The jobs executed on the local heterogeneous cluster. Error bars show the min/max of 3 runs.

special-purpose graph processing back-end (e.g., PowerGraph), even though these systems use fewer machines (PowerGraph runs on two machines versus Lindi which uses a seven-machine Naiad deployment). This is the case because general-purpose systems (like Hadoop MapReduce) work well for the batch phase of the workflow, but cannot execute the iterative PageRank as fast as specialised graph processing system, or general systems that use the timely dataflow model (i.e., Naiad).

However, a combination of Lindi and GraphLINQ, which both run jobs on Naiad, works best. This combination outperforms Lindi because it takes advantage of GraphLINQ's graph specific optimisations, and it outperforms Hadoop and PowerGraph because it avoids the extra I/O that results from moving intermediate data across back-end boundaries. Musketeer currently does not fully automatically generate the low-level Naiad code to combine Lindi and GraphLINQ. However, it could be easily extended to do so.

Musketeer's ability to flexibly partition a workflow makes it easy to explore different combinations of systems. In this section, I have shown that this can in some cases reduce workflow makespan (e.g., when using Lindi and GraphLINQ) or in other cases improve resource efficiency (e.g., when using Hadoop MapReduce and PowerGraph).

4.6 Automated mapping

There are many data processing systems, but no system always outperforms the others (§2.2.3). Developers find it difficult to know before executing their workflows which system is "best". In Chapter 3, I argued that this problem can be mitigated by building a workflow manager that allows developers to execute their high-level workflows on any back-end execution. Musketeer achieves this by decoupling front-end frameworks from back-end execution engines via an intermediate representation. Musketeer's developers can manually specify which back-end to use,

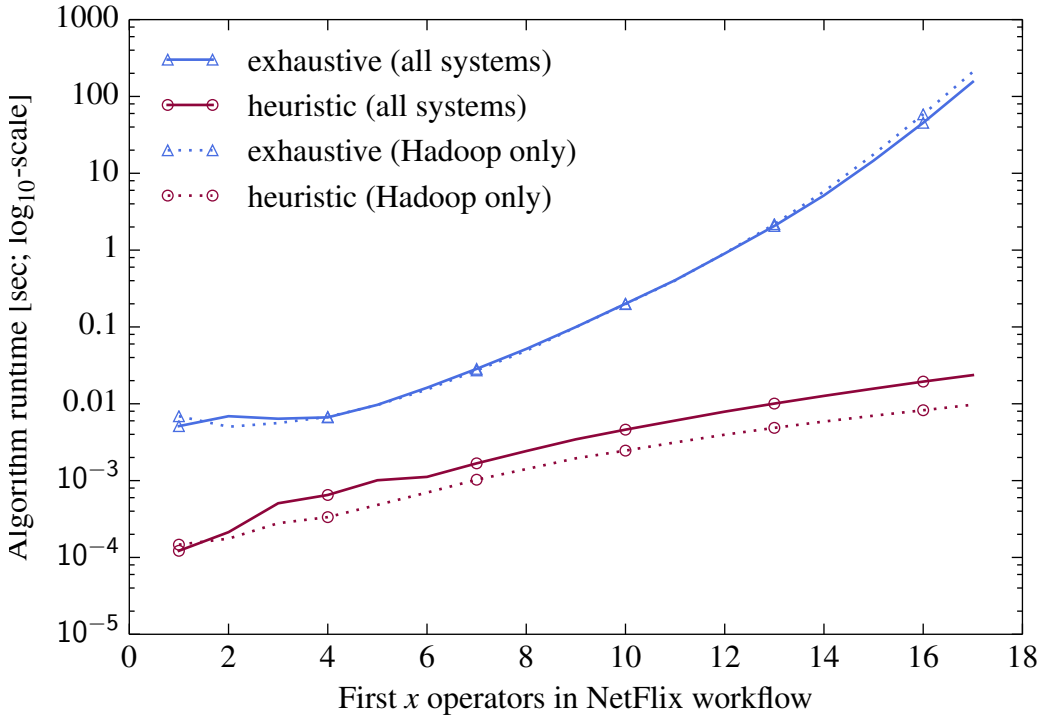


Figure 4.9: Runtime of Musketeer's DAG partitioning algorithms when considering the first x operators of an extended version of the Netflix workflow (N.B.: $\log_{10}$-scale y-axis).

but more importantly, they can let Musketeer automatically map their workflows to back-ends (§3.5).

In this section, I focus on evaluating Musketeer's scheduler for automatically mapping workflows. In order for the scheduler to meet developer requirements, it has to: *(i)* find mappings fast enough that it does not significantly increase workflow makespan, and *(ii)* choose mappings that are almost as good as the best options, or at least as good as the mappings developers would have chosen.

4.6.1 Runtime of Musketeer's automated mapping

First, I focus on Musketeer's DAG partitioning algorithms (§3.5.2). I measure the time it takes the exhaustive search and dynamic heuristic algorithms to partition the IR operator DAG. Ideally, they should not noticeably affect workflow makespan.

In the experiment, I measure partitioning runtime on workflows with an increasing number of operators. The workflows are subsets of an extended version of the Netflix workflow with a total of 18 operators. This workflows affords many operator merging opportunities, thus it makes a good and complex test case for the DAG partitioning algorithms.

In Figure 4.9, I show the runtimes for the two algorithms as the number of operators in a workflow increases. The exhaustive search has exponential complexity, but it manages to run in under a second for workflows with up 13 operators. However, beyond 13 operators, its runtime

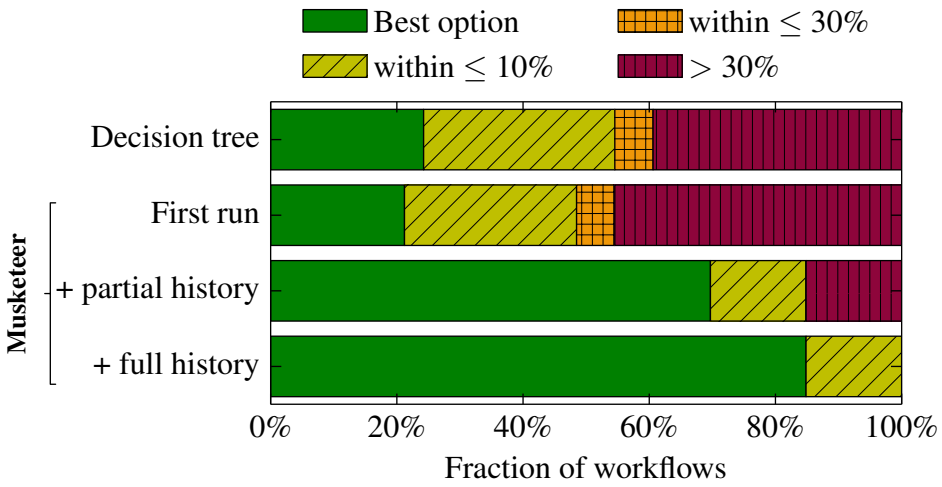


Figure 4.10: Makespan overhead of Musketeer's automated mapping choice compared to the best option. Workflow history helps, and Musketeer's cost function outperforms a simple decision tree.

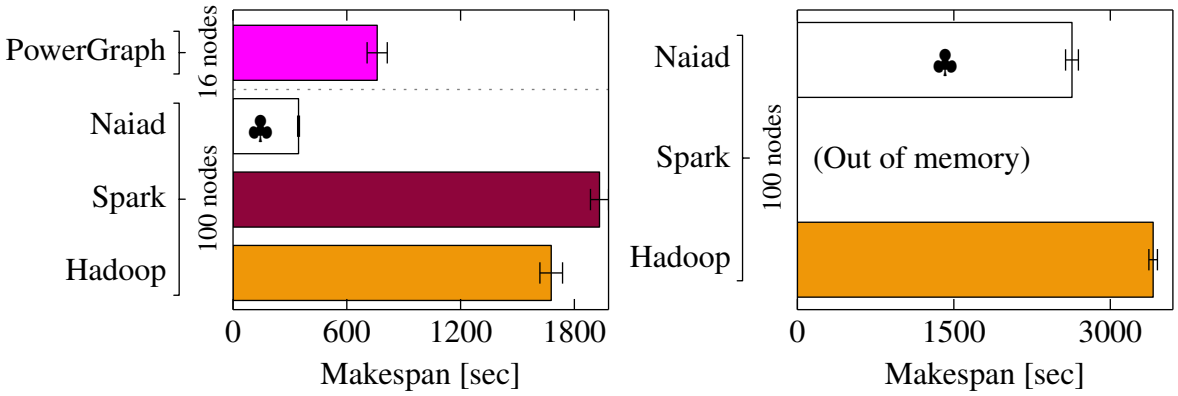
grows to more than 40 seconds. The exhaustive search guarantees that the optimal partitioning subject to the cost function is found. However, its runtime can quickly outweigh the makespan reduction resulted from executing the workflow with the optimal partitioning and mapping. In contrast, Musketeer's dynamic programming heuristic may not find the best partitioning, but it scales gracefully and runs in under 10ms even at 18 operators.

4.6.2 Quality of automated mapping decisions

Developers can manually map workflows to back-end execution engines, but great performance improvements can be made if Musketeer automatically chooses these mappings to back-ends. I now focus on evaluating the quality of Musketeer's automated mapping decisions (§3.5.1). First, I investigate decision quality using the workflows I previously executed (TPC-H query 17, top-shopper, Netflix movie recommendation, PageRank, Join, Project), and then I test decision quality on two additional workflows (single-source shortest path and *k*-means).

In the experiment, I use 33 different configurations by varying the input data sizes for the workflows. For each decision, I compare: *(i)* Musketeer's choice on the first run (with no workflow-specific history), *(ii)* its choice with incrementally acquired partial history, and *(iii)* the choice it makes when it has a full history of the per-operator intermediate data sizes. I also compare Musketeer's choices to those that emerge from a decision tree that Malte Schwarzkopf and I have developed based on our knowledge of data processing systems. The decision tree considers different back-ends features and known characteristics, and makes mappings accordingly (e.g., workflows with small inputs are executed on single-machine back-ends, graph analysis workflows are executed on specialised graph processing back-ends).

I consider a choice that achieves a makespan within 10% of the best option to be "good", and one within 30% as "reasonable". In Figure 4.10, I show the results: without any knowledge,



(a) Single-source shortest path (SSSP)

(b) *k*-means

Figure 4.11: Makespan of SSSP and *k*-means on the EC2 cluster (5 iterations). A club (♣) indicates Musketeer's choice.

Musketeer chooses good or optimal back-ends in about 50% of the cases. When partial workflow history is available, over 80% of its choices are good. Musketeer always makes good or optimal choices, if each workflow is initially executed operator-by-operator for profiling. By contrast, using the decision tree yields many poor choices. This is the case because the decision tree chooses on which back-end to run workflows using simple fixed thresholds based on input data size. It statically makes decisions ahead of workflow runtime, it does not adjust it decisions at runtime based on intermediate data sizes, and it is unable to accurately predict the benefits of operator merging and shared scans.

I also evaluate Musketeer's automatic mapping decisions on two new workflows: single-source shortest path (SSSP) and *k*-means clustering. SSSP can be expressed in vertex-centric systems, while *k*-means cannot. In Figure 4.11, I show workflow makespan for different back-ends and Musketeer's automated choice. The SSSP workflow receives as input the Twitter graph extended with costs, and I used 100M random points for *k*-means (100 clusters, two dimensions) [3]. Despite using my simple proof-of-concept cost function and a small training set, Musketeer correctly identifies the appropriate back-end (Naiad) in both cases.

In conclusion, Musketeer's automatic execution engine mapping solution makes good choices on the workflows I tested on. However, I only tested it in well-controlled environments in which only one workflow is running at a time. Musketeer's scheduler and cost function would have to be further refined in order to make good decisions in shared cluster environments. I discuss these challenges and future extensions in Chapter 7.

[3] My *k*-means uses the `CROSS JOIN` operator, which is inefficient. By replacing it, I could reduce the makespan and address Spark's OOM condition. However, I am only interested in the automated mapping here.

4.7 Summary

In this chapter, I have investigated Musketeer's ability to efficiently run data processing computations using a range of real-world workflows. My experiments show that Musketeer:

1. **Generates efficient code:** compared to time-consuming, hand-tuned implementations, Musketeer's automatically generated code is only up to 30% slower, yet Musketeer offers superior portability (§4.2, §4.3).

2. **Speeds legacy workflows:** Musketeer reduces legacy workflows' makespan by up to 2× by mapping them to a different back-end execution engine (§4.4).

3. **Flexibly combines back-ends:** by exploring combinations of multiple execution engines for a workflow, Musketeer finds combinations that outperform all single back-end alternatives (§4.5).

4. **Makes good automatic system mappings:** my automated mapping heuristic prototype makes good choices based on simple parameters that characterise execution engines (§4.6).

Although these results are encouraging, they must not be considered to hold for all versions of the data processing back-ends I use. Many of the performance issues I have highlighted could be addressed in future releases (e.g., Lindi's non-associative `GROUP BY`). However, new performance corner cases could also be introduced as system developers optimise for particular use cases. Musketeer is well placed to automatically discover these performance corner cases and to bring real benefits to developers by decoupling front-end frameworks and back-execution engines, and executing the workflows in the most appropriate combination of back-ends. Nonetheless, I believe my work represents only the first step in this promising direction. In Chapter 7, I discuss how Musketeer could be improved, and describe some future challenges.

Chapter 5

Firmament: a scalable, centralised scheduler

Modern data center clusters comprise of heterogeneous hardware and execute diverse workloads. These workloads consist of a range of tasks: from short-running interactive tasks that must complete within seconds to long-running service tasks that must meet service level objectives (SLOs). This task diversity and hardware heterogeneity make it challenging for cluster schedulers to achieve high utilisation in increasingly larger clusters, while keeping tasks completion times within seconds for interactive tasks, and meeting SLOs for service tasks. Cluster schedulers must satisfy the following three requirements to efficiently schedule the workloads from nowadays' clusters:

- *take into account hardware heterogeneity* and tasks' hardware preferences in order to choose quality task placements that reduce runtime or improve resource utilisation efficiency;
- *avoid co-locating interfering tasks* in order to meet service task SLOs and to reduce batch task runtimes;
- *choose task placements with low latency* at scale to ensure that interactive tasks complete within seconds, to quickly respond to new cluster events, and to keep resources highly utilised.

Both centralised and distributed state-of-the-art cluster schedulers strive to meet all three requirements, but in practice they fall short. On the one hand, centralised schedulers use complex algorithms that take into account hardware heterogeneity and avoid co-locating interfering tasks. But they take seconds or minutes to place tasks at scale and thus, do not meet the placement latency requirements of short-running interactive tasks [SKA+13; DSK15] (see §2.3.1.8). On the other hand, distributed schedulers use simple algorithms to place tasks with low scheduling latency at scale, but do not choose quality placements because they do not take into account hardware heterogeneity and task co-location interference [OWZ+13; RKK+16].

One of the main contributions of this dissertation is to show that centralised cluster schedulers can choose high-quality placements with low latency at scale. In this chapter, I extend Firmament [Sch16, §5], a min-cost flow-based scheduler, and show that with my extensions Firmament meets all the above-mentioned requirements. It maintains the same high placement quality as state-of-the-art centralised schedulers and matches the placement latency of distributed schedulers in the common case. Moreover, Firmament's placement latency degrades gracefully even in extreme situations when jobs comprise of tens of thousands of tasks or when clusters are oversubscribed.

Firmament uses the min-cost flow-based scheduling approach introduced by Quincy [IPC+09], but my work makes it scalable and generalises the approach. In Section 5.1, I give a high-level description of how min-cost flow schedulers work and how they differ from traditional task-by-task schedulers. In Section 5.2, I introduce Firmament's architecture and describe its flexible min-cost flow-based interface for scheduling. Following, I describe Flowlessly, a novel min-cost flow solver I developed for Firmament to offer low task placement latency at scale (§5.3). In Section 5.4, I describe several extensions to min-cost flow-based scheduling that developed and use one of them to build a scheduling policy that avoids task interference on cluster networks (§5.5). Finally, in Section 5.6, I discuss Firmament's limitations and how these could be addressed in future work.

5.1 Background

In this section, I give a high-level description of how min-cost flow-based schedulers work and discuss what distinguishes them most from other types of schedulers. I also introduce Quincy's scheduling policy which I use in the following sections to evaluate the techniques I developed to reduce Firmament's placement latency.

5.1.1 Task-by-task schedulers

In Section 2.3, I categorise cluster schedulers by the type architecture they use. However, cluster schedulers can also be categorised by how they process submitted tasks. Most cluster schedulers, whether centralised or distributed, are *queue-based* and place tasks one by one. In Figure 5.1, I show the stages through which a submitted task transitions in a task-by-task queue-based scheduler.

Task-by-task schedulers first add submitted tasks to a queue of unscheduled tasks. Following, they dequeue tasks *one by one*. For each task they perform a *feasibility check* to identify suitable machines, then *score* them according to their suitability, and finally *place* the task on the best-scoring machine. Scoring, i.e., rating the different placement choices for a task, can be expensive, and typically dominates scheduler algorithm runtime on large clusters. Cluster

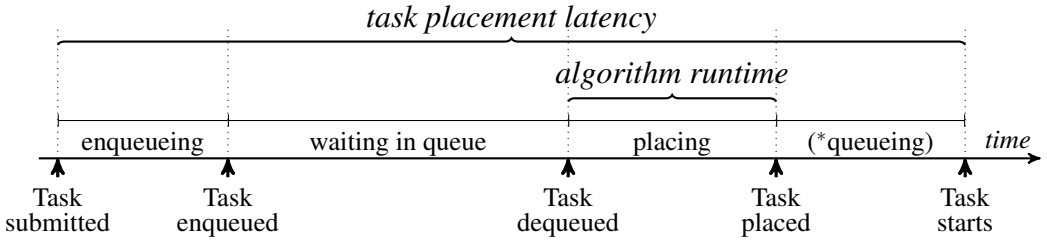


Figure 5.1: Stages a task proceeds through in task-by-task queue-based schedulers.

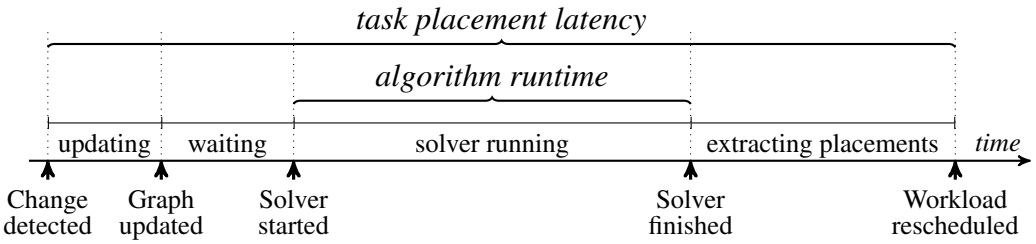


Figure 5.2: Stages min-cost flow-based schedulers proceed through.

schedulers use different techniques to keep scoring tractable. For example, Google's centralised Borg scheduler relies on several caching and approximation optimisations [VPK+15, §3.4]. The Sparrow distributed scheduler takes a more radical approach and does not score machines. Instead, it uses batch sampling to randomly select machines to dispatch reservations for a job's tasks. Task reservations are stored in "worker-side" queues, and tasks are only placed on machines when one of their reservations is at the front of a queue. Other schedulers also use "worker-side" queues [OWZ+13; BEL+14; RKK+16] to which one or more schedulers add tasks to [OWZ+13; DDK+15; DSK15; DDD+16].

Task-by-task queue-based schedulers have a fundamental limitation: they cannot consider how a task placement affects the placement options of the other queued tasks. Consider, for example, a scenario in which a cluster has available only one machine with a GPU. In the scheduler's queue there are two machine learning tasks among many others. One is at the front of the queue and has a preference for running on the machine with the GPU. The other one is towards the end of the queue and has a stronger preference than the first task for running on the same machine. A task-by-task queue-based scheduler places the first task on the machine with the GPU. However, the scheduler is next faced with two sub-optimal options when placing the second machine learning task: *(i)* assign the task to a sub-optimal machine or, *(ii)* migrate the first task to another machine – potentially loosing all the work the task has done – and place the second task on the machine. This is a fundamental limitation, task-by-task schedulers increase task makespan and cause work to be wasted.

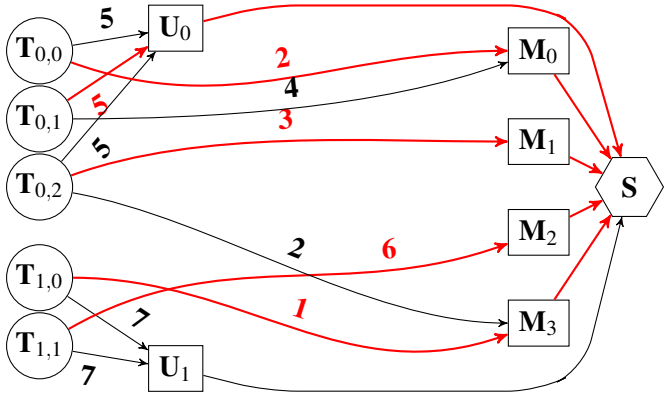


Figure 5.3: Flow network for a four-machine cluster with two jobs of three and two tasks. All tasks except $\mathbf{T}_{0,1}$ are scheduled on machines. Arc labels show non-zero cost, and those with flow are in red. All arcs have unit capacity, and the red arcs form the min-cost solution.

5.1.2 Min-cost flow-based schedulers

An alternative approach only suitable for centralised schedulers is *min-cost flow-based* scheduling, introduced by Quincy [IPC+09]. This approach uses a placement mechanism – min-cost flow optimisation – with an attractive property: it guarantees overall optimal task placements for a given scheduling policy. In contrast to task-by-task schedulers, a flow-based scheduler not only schedules new tasks, but also reconsiders the entire existing workload ("rescheduling"), preempting and migrating tasks if prudent.

In Figure 5.2, I show the stages a min-cost flow-based scheduler transitions through when scheduling workloads. When a change to the cluster state occurs (e.g., task submission, task failure), the scheduler updates its internal graph-based representation of the cluster and the scheduling problem. Next, the scheduler runs a min-cost flow optimisation over its internal graph-based representation. The min-cost flow optimisation yields an optimal minimum cost flow from which the scheduler extracts task placements. If changes occur to the cluster state while the scheduler executes the optimisation, the scheduler updates its internal graph, but only runs a new optimisation once the prior optimisation completes.

Min-cost flow-based schedulers use flow networks for their internal graph-based representation of the cluster state and the scheduling problem. A flow network is a directed graph whose arcs carry *flow* from source nodes to a sink node. A *cost* and *capacity* associated with each arc constrain the flow, and specify preferential routes for it.

In Figure 5.3, I show an example of a flow network that expresses a simple cluster scheduling problem. Each task node $\mathbf{T}_{j,i}$ on the left hand side, represents the i^{th} task of job j. Each task node is a source of one unit of flow. All flow must be drained into the sink node ($\mathbf{S}$) for a feasible solution to the optimisation problem. To reach $\mathbf{S}$, flow from $\mathbf{T}_{j,i}$ can proceed through a machine node ($\mathbf{M}_m$), which schedules the task on machine m (e.g., $\mathbf{T}_{0,2}$ on $\mathbf{M}_1$). Alternatively, the flow may proceed through a special "unscheduled aggregator" node ($\mathbf{U}_j$ for job j) to the sink, and consequently leave the task unscheduled (e.g., $\mathbf{T}_{0,1}$).

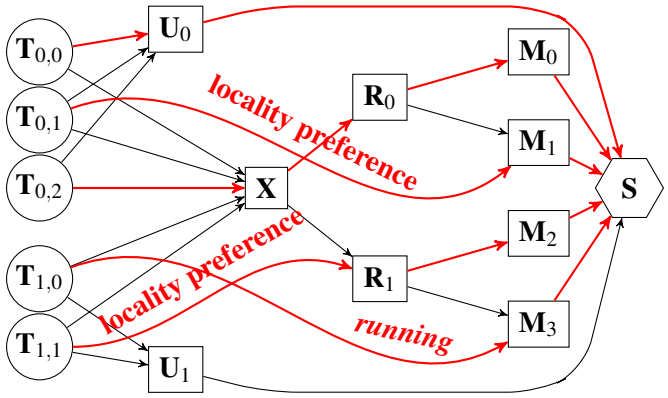


Figure 5.4: Adding aggregators to the flow network in Figure 5.3 enables different scheduling policies. The example shows the Quincy [IPC+09] policy with cluster (**X**) and rack (**R**) aggregators.

In Figure 5.3, a task's placement preferences are expressed as costs on direct arcs to machines (e.g., 2 for placing $\mathbf{T}_{0,2}$ on $\mathbf{M}_3$). The cost to leave the task unscheduled – or to preempt it when running – is the cost on its arc to the unscheduled aggregator (e.g., 7 for $\mathbf{T}_{1,1}$). Given this flow network, a min-cost flow solver finds a globally optimal (i.e., minimum-cost) task placement (flow highligthed in red in Figure 5.3). Task placements are extracted from this flow by tracing flow paths from the machine nodes back to the task nodes.

In the example, the optimal flow expresses the best trade-off between tasks' unscheduled wait time and their placement preferences. The optimization places tasks with strong preference (i.e., low arc cost to machine nodes), and leaves unscheduled tasks that can wait until resources are available (i.e., low arc cost to unscheduled aggregator nodes).

In the example, task nodes are directly connected to machine nodes. For a flow network with such a structure, a min-cost flow-based scheduler is guaranteed to find the overall optimal placement with regards to the arcs' cost if each task node is connected to each machine node. But this requires billions of arcs to be added to a large cluster's flow network. The scheduler would take minutes to run the optimisation on such a large network. However, the optimal placements can also be found without connecting each task node directly to each machine node. Arcs can connect tasks to aggregator nodes, similar to the unscheduled aggregators. Such aggregators may, for example, group machines in a rack, or similar tasks (e.g., tasks in a particular job, tasks with the same resource requirements). The aggregator nodes allow different scheduling policies to be expressed without requiring an inordinate number of arcs. When using aggregators, the cost of a task placement option is the sum of all arc costs on the path to the sink.

In Figure 5.4, I illustrate this idea with the original *Quincy scheduling policy* [IPC+09, §4.2]. This policy is designed for batch jobs, and optimises for a trade-off between data locality, task unscheduled wait time, and task preemption cost. The Quincy policy uses rack aggregators ($\mathbf{R}_r$) to group machines that share racks and a cluster-level aggregator (**X**) to group racks. Tasks have low-cost *preference* arcs to machines and racks on which they achieve high data locality, but fall

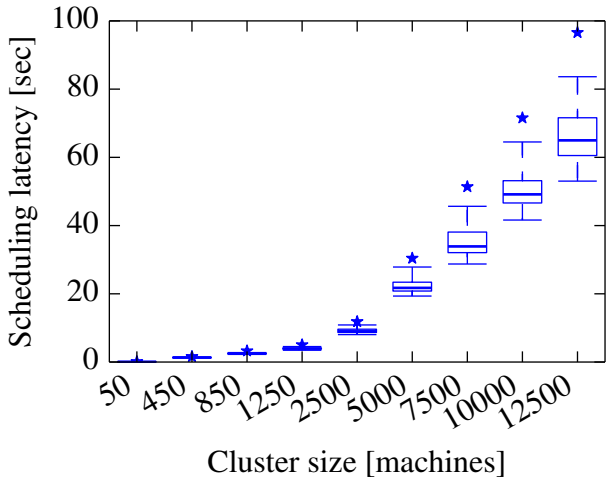


Figure 5.5: Quincy [IPC+09] scales poorly as cluster size grows. Simulation on subsets of the Google trace; boxes are 25th, 50th, and 75th percentile delays, whiskers 1st and 99th, and a star indicates the maximum value.

back to higher-cost arcs connected to the cluster aggregator if none of their preferred resources is available (e.g., $\mathbf{T}_{0,2}$).

However, min-cost flow-based schedulers do not choose placements with low-latency even when they use policies that take advantage of aggregator nodes. In Figure 5.5, I show that scheduling latency is too high to place interactive tasks at scale when using the Quincy policy and the current state-of-the-art min-cost flow solver (i.e., cost scaling [Gol97]). In the experiment, I replay subsets of a public trace of one of Google's clusters [RTG+12], which I augment with locality preferences for batch processing jobs[1] against my implementation of the Quincy scheduling policy. I measure the scheduler algorithm runtime for clusters of increasing size. The placement latency increases with scale, up to a median of 64s and a 99th percentile of 83s for the full Google cluster (12,500 machines). During this time, the scheduler must wait for the optimisation to finish, and cannot make placements for any newly submitted tasks. Moreover, tasks may finish and free resources, but the scheduler cannot replace the tasks with new ones and thus, wastes resources.

In this experiment and in general, the min-cost flow optimisation algorithm runtime dominates the scheduling latency. In this dissertation, I describe several techniques that I developed to reduce min-cost flow solvers' runtime and scheduling latency. Furthermore, I extend min-cost flow-based schedulers with new features (e.g., complex constraints, convex arc costs) that can be used to improve placement quality.

[1]Details of my simulation are in §6.1; in the steady-state, the 12,500-machine cluster runs about 150,000 tasks comprising about 1,800 jobs.

5.2 Firmament overview

Firmament, like Quincy, models the scheduling problem as a min-cost flow optimisation over a *flow network*. I chose to extend the Firmament min-cost flow-based scheduler for three reasons. First, flow-based schedulers consider entire workloads, and thus can support rescheduling and priority preemption. Second, flow-based schedulers achieve high placement quality and, consequently, low workflow makespan [IPC+09, §6]. Third, flow-based schedulers amortise work well over many task placements, and hence achieve high task throughput – albeit at a high placement latency that I show it can be reduced.

Why my extensions, the Firmament scheduler offers three key benefits over the state-of-the-art Quincy min-cost flow scheduler:

1. It makes placements at **low, sub-second latency** at scale, sufficiently fast to support interactive and short-running tasks.
2. It provides a flexible API that can be used to express many **different scheduling policies**.
3. It supports **complex scheduling features** that previously could not be expressed in min-cost flow-based schedulers (e.g., gang-scheduling) or were thought to be too expensive (e.g., resource hogging avoidance).

Firmament has a flexible design and can execute as a centralised, a hierarchical distributed or a fully distributed scheduler architecture. Each machine in the cluster runs a Firmament coordinator process. Coordinators schedule tasks, monitor tasks, and collect performance counter information and task resource utilisation statistics. Firmament arranges coordinators in a hierarchy tree of coordinators, in which each coordinator schedules tasks on the machine it runs onto or delegates tasks to child coordinators. Schwarzkopf describes in detail how Firmament hierarchically groups coordinators and uses optimistically concurrent shared-state to support the above-mentioned architectures [Sch16, §5.3].

In this dissertation, I focus on reducing task placement latency and improving placement quality of min-cost flow-based schedulers. Unlike other types of schedulers, min-cost flow-based schedulers reconsider the entire workload (running and waiting tasks) each time they schedule tasks. They achieve highest placement quality when they have up-to-date information about the entire workload and about resource utilisation on each cluster machine. Thus, in this dissertation, I configure Firmament to run a single centralised cluster scheduler that aggregates information about the entire cluster.

In Figure 5.6, I give an overview of the Firmament scheduler architecture I use in my work. All tasks are submitted to the "master coordinator" process which schedules and delegates them to worker coordinator processes that run on worker machines. The worker coordinators are simple task executors because they use no-op schedulers to place delegated tasks (i.e., they abide to the placement decisions made by the master coordinator).

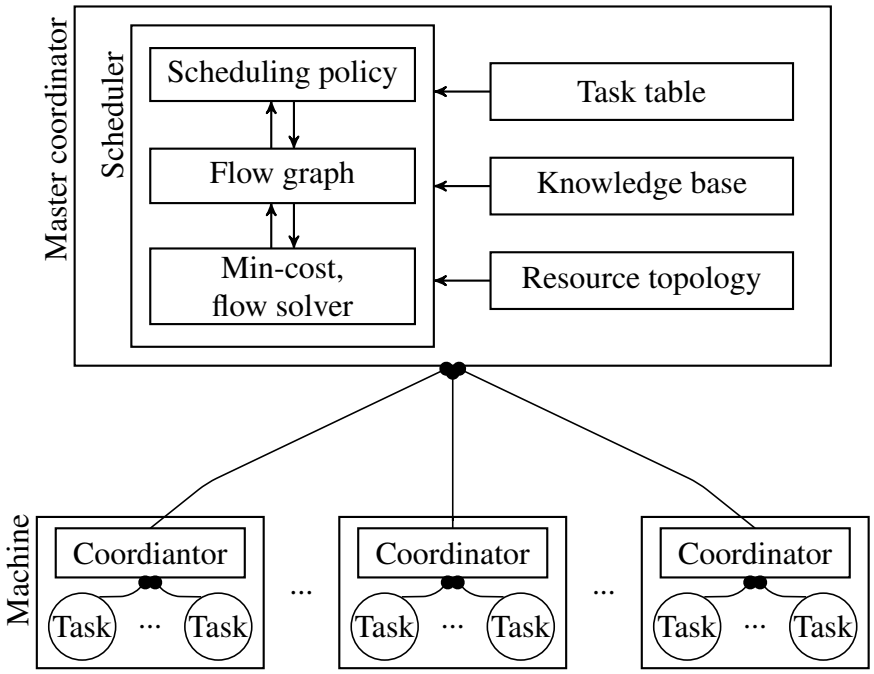


Figure 5.6: Firmament's scheduling policy modifies the flow network according to workload, cluster, and monitoring data; the flow network is passed to the min-cost flow solver, whose computed optimal flow yields task placements.

Upon start-up, each worker coordinator extracts the micro-architectural topology of the machine on which it runs, and submits the topology to the master coordinator. The master combines each machine's micro-architectural topology into a larger *resource topology* that can include information about network topology and how machines are grouped into racks. Worker coordinators also automatically collect performance counter information and resource utilisation statistics of running tasks. They send these statistics to the master coordinator which aggregates and stores them in its *knowledge base*.

Firmament also stores in the knowledge base task-specific information. Many data center workflows run periodically [AKB+12b], and tasks executed by different instances of a periodic workflow have similar resource usage patterns. For such tasks, Firmament constructs profiles that contain information about tasks' suitability to run on different types of hardware and about key characteristics (e.g., cache working set size, network bandwidth usage). The master coordinator's scheduler can use the task profiles and the cluster resource topology in scheduling policies to avoid task co-location interference and to take into account hardware heterogeneity when choosing placements.

Firmament generalises min-cost flow-based scheduling over the single, batch-oriented policy proposed by Quincy. Cluster administrators can use a policy API to configure Firmament's *scheduling policy*, which for example, may incorporate multi-dimensional resources, fairness, and priority preemption [Sch16, Appendix C]. The scheduling policy defines a *flow network* that models the cluster using nodes to represent tasks and machine micro-architectural topologies. The policy can also use task profiles to encode tasks' preferences for particular resources in

the flow network, and to give hints on where tasks should be placed to reduce interference. Following, Firmament submits the flow network to a *min-cost flow solver* that finds an optimal (i.e., min-cost) flow. Finally, after the solver completes, Firmament extracts the implied task placements from the optimal flow.

Firmament continuously monitors cluster events (e.g., task completions, machine failures), and updates the flow network. However, Firmament cannot react to these events while the solver runs because min-cost solver are not incremental. Yet, as soon as the solver completes, Firmament modifies the graph according to its scheduling policy in response to monitoring information and events that occurred while the solver was running. Following, Firmament reruns the solver to compute the new optimal flow. In a busy cluster (i.e., with many task and machine events), the solver is executed almost continuously.

My goal is to improve Firmament such that it makes high-quality scheduling placements at the lowest possible latency. This requires me to efficiently solve an algorithmically challenging min-cost flow problem. To achieve this, I study several min-cost flow optimisation algorithms and their performance (§5.3.1). The key insight I discover is that even centralised sophisticated min-cost flow algorithms for the scheduling problem can be fast: *(i)* if they match the problem structure well, and *(ii)* if few changes to cluster state occur while the algorithm runs.

Based on this insight I have implemented Flowlessly, a new min-cost flow solver for Firmament which supports four algorithms that I describe in §5.3.1. Each algorithm has edge cases when it fails to place tasks with low latency. In Section 5.3, I investigate algorithms' properties and three techniques to reduce Firmament's placement latency: non-optimal approximate flow solutions, incremental flow re-optimisation and problem-specific heuristics. Flowlessly leverages these techniques to quickly compute the optimal flow. Flowlessly consists of 8,000 lines of C++ code, while Firmament's underlying cluster manager and my simulator – that I use to evaluate Flowlessly – are implemented in 24,000 lines of C++.

5.3 Flowlessly: a fast min-cost flow solver

In order for Flowlessly to quickly find the optimal flow for large flow networks I had to overcome four challenges:

1. Algorithms for min-cost flow that are slow when faced with many changes can work extremely well when there are only a few: *I had to devise a way to return the flow as fast as the best-suited algorithm.*

2. Efficiently switching between algorithms while maintaining graph state is tricky because algorirthms require different invariants to hold: *I had to combine insights from multiple algorithms to achieve low-overhead transitions between algorithms.*

3. Standard implementations of min-cost flow algorithms can underperform on the type of flow graphs Firmament generates: *I had to develop problem-specific heuristics to reduce the runtime of the algorithms*.

4. Existing algorithms for min-cost flow do not maintain state between runs or support incremental re-optimisation: *I had to adjust several min-cost flow algorithms to work incrementally*.

In this section, I survey several min-cost flow optimisation algorithms and demonstrate that choosing an algorithm that is suitable for the flow network's structure, but does not have the best worst case complexity, yields substantial runtime reductions (§5.3.1). Approximate solutions might be sufficient for the scheduling problem. Thus, I consider if solver runtime reductions are achievable with approximate min-cost flow solutions, but find that they generate unacceptably poor and volatile placements (§5.3.3). Next, I investigate which min-cost flow algorithms are amenable to running incrementally, re-optimising a previous solution (§5.3.4). Following, I discuss two heuristics that are specific to the scheduling problem and the types of graphs it generates, but reduce runtime of min-cost flow algorithms (§5.3.5). I combine these algorithmic insights with several implementation-level techniques to further reduce Flowlessly's runtime. Flowlessly runs two min-cost flow algorithms concurrently to avoid slowdown in edge cases (§5.3.6). Finally, I discuss two novel algorithms I developed to optimise Flowlessly's interaction with Firmament. These algorithms efficiently update the flow network in response to cluster state changes and quickly extract task placements from the computed optimal flow (§5.3.7).

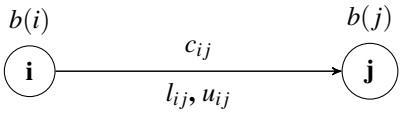
5.3.1 Min-cost flow algorithms

A min-cost flow algorithm takes a directed flow network $G = (N, A)$ as input. Each arc $(i, j) \in A$ has a cost c_{ij}, a minimum flow requirement l_{ij} and a maximum flow capacity u_{ij} (see Figure 5.7a). Moreover, each flow network node $i \in N$ has an associated supply $b(i)$; nodes with positive supply are *sources*, those with negative supply are *sinks*.

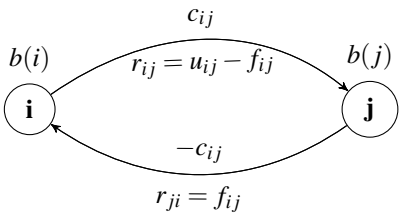
Informally, min-cost flow algorithms must optimally (i.e., with smallest cost) route the flow from all sources (i.e., $b(i) > 0$) to sinks (i.e., $b(i) < 0$) meeting the minimum flow requirements and without exceeding the maximum flow capacity on any arc. For example, for networks generated using the Quincy scheduling policy (see Figure 5.4) the flow must be routed from task nodes ($\mathbf{T}_{i,j}$) to the sink node (**S**), which has a flow demand equal to the total number of tasks.

To understand the differences between min-cost flow algorithms, I introduce a a more formal definition: the goal of a min-cost flow algorithm is to find a flow f that minimises Eq. 5.1, while respecting the flow *feasibility constraints* of **mass balance** (Eq. 5.2) and **capacity** (Eq. 5.3):

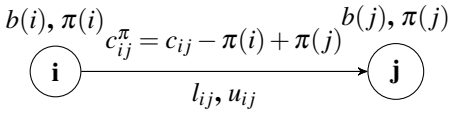
$$\text{Minimise} \sum_{(i,j)\in A} c_{ij} f_{ij} \text{ subject to} \tag{5.1}$$



(a) Each flow arc is directed and has associated a cost c_{ij}, a minimum flow requirement l_{ij} and a maximum capacity u_{ij}.



(b) In the residual network each arc (i,j) has a residual capacity $r_{ij} = u_{ij} - f_{ij}$ and a reverse arc (j,i) with $r_{ji} = f_{ij}$ and $c_{ji} = -c_{ij}$.



(c) The reduced cost of a flow arc is the sum between its cost and the difference of its node potentials.

Figure 5.7: Examples of different types of flow network arcs.

$$\sum_{k:(j,k)\in A} f_{jk} - \sum_{i:(i,j)\in A} f_{ij} = b(j), \forall j \in N \tag{5.2}$$

$$\text{and } l_{ij} \leq f_{ij} \leq u_{ij}, \forall (i,j) \in A \tag{5.3}$$

A flow that satisfies the capacity constraints but not the mass balance constraints is called a *pseudoflow*.

Some algorithms use an equivalent definition of the flow network called *residual network* ($G'(f)$). In the residual network, each arc $(i,j) \in A$ with a cost c_{ij}, a l_{ij} flow requirement and maximum capacity u_{ij} is replaced by two arcs: (i,j) and (j,i). Arc (i,j) has cost c_{ij}, a flow requirement of $l'_{ij} = max(l_{ij} - f_{ij}, 0)$, and a *residual capacity* of $r_{ij} = u_{ij} - f_{ij}$. Arc (j,i) has cost $-c_{ij}$, a zero flow requirement, and a *residual capacity* of $r_{ji} = f_{ij}$ (see Figure 5.7b). The feasibility constraints also apply in the residual network.

The *primal* minimisation problem (Eq. 5.1) also has an associated *dual* problem, which some algorithms solve more efficiently. In the dual min-cost flow problem, each node $i \in N$ has an associated dual variable $\pi(i)$ called *potential* (Figure 5.7c). Moreover, each arc has a *reduced cost* with respect to the node potentials, defined as:

$$c^{\pi}_{ij} = c_{ij} - \pi(i) + \pi(j), \forall (i,j) \in A \tag{5.4}$$

The reduced cost does not change the cost of any directed cycle W in the flow network because the following equality holds:

$$\sum_{(i,j)\in W} c_{ij}^{\pi} = \sum_{(i,j)\in W} (c_{ij} - \pi(i) + \pi(j)) = \sum_{(i,j)\in W} c_{ij} + \sum_{(i,j)\in W} (\pi(j) - \pi(i)) = \sum_{(i,j)\in W} c_{ij}$$

In the equality, $\sum_{(i,j)\in W} (\pi(j) - \pi(i))$ reduces to zero because each cycle node's potential $\pi(j)$ occurs first with a positive sign for arc (i, j) and with a negative sign for the next arc in the cycle (j,k). Similarly, for each directed path P from node i to node j the following equality holds for its reduced cost:

$$\sum_{(i,j)\in P} c_{ij}^{\pi} = \sum_{(i,j)\in P} (c_{ij} - \pi(i) + \pi(j)) = \sum_{(i,j)\in P} c_{ij} + \sum_{(i,j)\in P} (\pi(j) - \pi(i)) = \sum_{(i,j)\in P} c_{ij} - \pi(i) + \pi(j)$$

With the exception of node i and node j, all nodes in the path P occur as both a source and a destination of an arc. Thus, the contribution their potential makes to the reduced cost is zero.

The two equations from above can be used to show that the dual min-cost flow problem is equivalent to the primal problem [AMO93, §2.4]. In contrast to the primal problem that minimises $\sum_{(i,j)\in A} c_{ij} f_{ij}$, the goal of the dual problem is to:

$$\text{Maximise} \sum_{i\in N} b(i) * \pi(i) + \sum_{(i,j)\in A} c_{ij}^{\pi} f_{ij} \tag{5.5}$$

subject to the flow feasibility constraints of **mass balance** (Eq. 5.2) and **capacity** (Eq. 5.3).

Regardless if a min-cost flow algorithm solves the primal or the dual problem, the algorithm completes when it finds an optimal feasible flow. A feasible flow is also optimal if and only if at least one of following three *optimality conditions* is satisfied.

Theorem 5.3.1. ***Negative cycle optimality conditions.*** *A feasible flow f is an optimal flow of minimum cost if and only if the residual network $G'(f)$ contains no directed negative-cost cycle.*

Intuitively, if f is a feasible flow and $G'(f)$ contains a directed negative-cost cycle then f cannot be optimal because a feasible flow with a smaller cost can be obtained by increasing the flow along the cycle.

Theorem 5.3.2. ***Reduced cost optimality conditions.*** *A feasible flow f is an optimal flow of minimum cost if and only if there exists a set of node potentials π such that there are no arcs in the residual network $G'(f)$ with negative reduced cost.*

Intuitively, if f is a feasible flow and $G'(f)$ contains only arcs (i,j) with $c^{\pi}_{ij} \geq 0$ then the reduced cost of every directed cycle W in $G(f)$ is $\sum_{(i,j)\in W} c^{\pi}_{ij} \geq 0$. The residual flow network $G'(f)$ does not contain any negative reduced cost cycle which entails that Theorem 5.3.1 holds and that f is optimal.

Theorem 5.3.3. ***Complementary slackness optimality conditions.*** *A feasible flow f is an optimal flow of minimum cost if and only if there is a set of node potentials π such that the reduced arc costs and flows satisfy the following conditions for every arc $(i,j) \in A$:*

$$\text{If } c^{\pi}_{ij} > 0, \text{ then } f_{ij} = l_{ij} \tag{5.6a}$$

$$\text{If } l_{ij} \leq f_{ij} \leq u_{ij}, \text{ then } c^{\pi}_{ij} = 0 \tag{5.6b}$$

$$\text{If } c^{\pi}_{ij} < 0, \text{ then } f_{ij} = u_{ij} \tag{5.6c}$$

Informally, if either of the complementary slackness conditions is not satisfied then one of the other optimality conditions is not satisfied as well. Consider the following three cases that cover all the possible reduced cost values for any arc $(i,j) \in A$:

- Case 1: If $c^{\pi}_{ij} > 0$ then $c^{\pi}_{ji} < 0$ according to the definition of the residual network. But $(j,i) \notin G'(f)$ because otherwise it would not respect the reduced cost optimality conditions. Therefore, the arc's flow must be equal to its minimum flow requirement (i.e., $f_{ij} = l_{ij}$).
- Case 2: If $l_{ij} \leq f_{ij} \leq u_{ij}$ then both (i,j) and $(j,i) \in G'(f)$. These inequalities and the reduced cost optimality conditions require that both $c^{\pi}_{ij} \geq 0$ and $c^{\pi}_{ji} \geq 0$. But, zero is the only value that respects both conditions because $c^{\pi}_{ij} = -c^{\pi}_{ji}$ in the residual network $G'(f)$.
- Case 3: If $c^{\pi}_{ij} < 0$ then arc $(i,j) \notin G'(f)$ because otherwise it would not respect the reduced cost optimality conditions. Therefore, the arc's flow must be equal to its capacity (i.e., $f_{ij} = u_{ij}$).

Substantial research effort has gone into min-cost flow algorithms. All the existing algorithms work as a series of iterations in which they either: *(i)* maintain flow feasibility and work to achieve optimality by finding a flow that respects one of above-mentioned types of optimality conditions or, *(ii)* refine a flow that respects the optimality conditions until the flow is feasible.

I now describe several competitive min-cost flow algorithms to lay the groundwork for the following sections in which I explain why I decided to implement four algorithms in Flowlessly, and how I optimise some of these algorithms.

Cycle canceling. The simplest min-cost flow algorithm is **cycle canceling** [Kle67]. The algorithm first finds a feasible flow using any max-flow algorithm. Following, it performs a series of iterations, during which it maintains flow feasibility and attempts to achieve optimality. In each iteration, cycle canceling augments flow along negative-cost directed cycles in the residual graph. Pushing flow along such a cycle guarantees that the overall solution cost decreases. The algorithm finds a feasifle optimal flow when no negative-cost cycles exist in the graph (i.e., the negative cycle optimality conditions are met).

Minimum mean cycle canceling. Minimum mean cycle canceling, like cycle canceling, first uses a max-flow algorithm to find a feasible flow, and then it runs a series of iterations during which it maintains flow feasibility [GT89]. In contrast to cycle canceling, minimum mean cycle canceling augments flow along cycle W with the smallest mean cost in the residual network rather than any negative-cost cycle. The algorithm completes when there are no cycles with a mean negative cost left in the flow network. The flow is feasible and it satisfies the negative cycle optimality conditions. Unlike cycle canceling, which is a weakly polynomial-time algorithm, minimum mean cycle canceling is a strongly polynomial-time algorithm because its runtime depends only on the number of nodes and arcs in the flow network. However, the algorithm is unlikely to outpeform cycle canceling on the flow networks typically generated by Firmament's policies (i.e., in which arc capacities and costs are much smaller than the number of nodes in the network).

Successive shortest path. Unlike the above-mentioned algorithms, the successive shortest path [AMO93, p. 320] algorithm does not maintain a feasible flow at each step. Instead, the algorithm maintains a pseudoflow (a flow that satisfies the capacity constraints, but does not satisfy the mass balance constraints) and the reduced cost optimality conditions at each step. The algorithm iteratively refines the pseudoflow until it satisfies the mass balance constraints and thus becomes feasible. Successive shortest path repeatedly selects a source node (i.e., $b(i) > 0$), finds the path with the smallest cost in the residual network from it to the sink, and sends flow along this path. The algorithm completes with an optimal flow when there are no source nodes left in the flow network.

Primal-dual. Like successive shortest path algorithm, the primal-dual algorithm maintains a pseudoflow that satisfies the reduced cost optimality conditions at each step [FF57]. The algorithm transforms the network flow $G = (N,A)$ to an equivalent single source and single sink network. It first introduces a new source node *src* with a supply equal to the sum of supplies of prior source nodes (i.e., $b(src) = \sum_{b(i)>0} b(i)$). The algorithm connects each node i with $b(i) > 0$ to the new source node *src* with with zero cost and $b(i)$ capacity arcs (src, i). Similarly, the algorithm introduces a new sink node *dst* with a supply $b(dst) = \sum_{b(i)<0} b(i)$ to which it connects all prior sink nodes with zero cost and $-b(i)$ capacity arcs. Next, the algorithm transforms the

graph to only contain a supply *src* node and a sink *dst* node by setting $b(i) = 0$ for all nodes $i \in N$.

Unlike successive shortest path, which iteratively augments flow along the path with the smallest cost, primal-dual uses reduced costs and simultaneously augments flow from the source node *src* to the sink node *dst* along several paths. In each iteration, the algorithm computes for each node i the smallest reduced cost $d(i)$ to reach the node i from the source node *src*. Following, it subtracts $d(i)$ from $\pi(i)$ for each node i. This creates at least a path of zero reduced cost arcs from the source node *src* to the sink node *dst*. Thus, primal-dual can simultaneously augment flow along several paths zero reduced cost paths by computing a maximum flow on residual network's zero reduced cost subgraph, called the admissible network $G^{\circ}(f) = (N \cup \{src, dst\}, A^{\circ})$. Each such step reduces the supply at the source node *src* (i.e., improves feasibility) without breaking the reduced cost optimality conditions (i.e., pseudoflow optimality is maintained). Primal-dual completes with an optimal feasible flow when it completes pushing the all the supply from the source node *src* to the sink node *dst*.

Out-of-kilter. Unlike successive previous two algorithms, the out-of-kilter algorithm maintains a flow that satisfies the mass balance constraints at each step, but the flow does not satisfy the capacity constraints and the optimality conditions [Ful61]. Out-of-kilter iteratively modifies arc flows and node potentials such that at each step it improves feasibility and reduces the overall flow cost. The algorithm divides arcs into two categories: *(i)* arcs that satisfy the complementary slackness optimality conditions (i.e., in-kilter arcs), and *(ii)* arcs that do not satisfy the conditions (i.e., out-of-kilter arcs). In each iteration, the algorithm transforms one or more out-of-kilter arcs into in-kilter arcs and ensures that all already in-kilter arcs continue to satisfy the complementary slackness optimality conditions.

In each iteration, the algorithm chooses an out-of-kilter arc (i, j) and computes the *shortest path distances d* from node j to all nodes. The shortest path distance between two nodes is the smallest sum of arc lengths of any path between the two nodes. The *arc length* of arc (p, q) is $max(0, c^{\pi}_{pq})$. Following, the subtracts $d(i)$ from $\pi(i)$ for each node i. If any updated reduced cost c^{π}_{ij} becomes negative then the residual network contains a negative-cost cycle W comprising of the shortest distance path from node j to node i and arc (i, j). The out-of-kilter algorithm augments flow along cycle W, and thus transforms the in-kilter (i, j) arc in an out-of-kilter arc. The algorithm completes with an optimal feasible flow when there are no in-kilter arcs left in the residual network.

Relaxation. The relaxation algorithm [BT88a; BT88b], like successive shortest path, maintains a flow that satisfies the reduced cost optimality conditions and each step, and augments flow from source nodes along the shortest path to the sink. However, unlike successive shortest path, relaxation optimises the dual problem by applying one of the following two changes when possible:

1. Keeping π unchanged, the algorithm modifies the flow, f, to f' such that f' still respects the reduced cost optimality condition and the total supply decreases (i.e., feasibility improves).

2. It modifies π to π' and f to f' such that f' is still a reduced cost-optimal solution and the cost of that solution decreases (i.e., total cost decreases).

This allows relaxation to decouple the improvements in feasibility from reductions in total cost. When relaxation has the possibility of reducing cost or improving feasibility, it chooses to reduce cost.

Capacity scaling. The successive shortest path and the relaxation algorithms might conduct many flow path augmentations that push small amounts of flow in some networks. In the worst case, successive shortest path can runs up to $N*U$ path augmentations, where N is the number of nodes and U is the largest arc capacity. By contrast, the capacity scaling algorithm is guaranteed to push a sufficiently large flow in each augmentation [EK72]. Capacity scaling conducts at most $M*log(U)$ augmentations, where M is the number of arcs.

Capacity scaling maintains a pseudoflow that satisfies the reduced cost optimality conditions. The algorithm converts the pseudoflow into an optimal flow in a series of Δ capacity scaling phases. In a Δ-scaling phase, the algorithm augments only paths that can carry exactly Δ units of flow. If there are no source nodes *src* with $b(src) \geq \Delta$, or no sink nodes *dst* with $b(src) \leq -\Delta$ left in the network, the algorithm halves Δ and starts another scaling phase. Capacity scaling completes when $\Delta = 1$ and there are no nodes with flow supply or demand left in the network. The resulted flow is optimal because there is no supply left (i.e., satisfies mass balance) and because it satisfies the reduced cost optimality conditions.

Cost scaling. Cost scaling [GT90; GK93; Gol97] iterates to reduce flow cost while maintaining feasibility. It uses a relaxed complementary slackness condition called ε-optimality. A flow is ε-optimal if the flow on arcs with $c_{ij}^{\pi} > \varepsilon$ is zero and there are no arcs with $c_{ij}^{\pi} < -\varepsilon$ on which flow can be sent. Initially, the algorithm starts with ε equal to the maximum arc cost. In each iteration, the algorithm relabels nodes (i.e., adjusts node potential) and pushes flow along arcs with $c_{ij}^{\pi} < -\varepsilon$ in order to achieve ε-optimality. Once the flow is ε-optimal, the algorithm divides ε by a constant factor and starts another iteration. Cost scaling completes when $\frac{1}{n}$-optimality is achieved, because this is equivalent to the complementary slackness optimality condition [Gol97].

Double scaling. The double scaling algorithm combines ideas from the capacity scaling and cost scaling algorithms [AGO+92]. Like cost scaling, double scaling refines ε in a series of iterations in which it achieves ε-optimality. However, instead of relabelling nodes and pushing flow from them, the double scaling algorithm uses Δ capacity scaling phases to achieve ε-optimality.

The algorithm completes when $\varepsilon = \frac{1}{n}$, $\Delta = 1$ and there are no suply and sink nodes left in the graph. Double scaling has a better worst-case complexity than each algorithm it combines has individually (Table 5.1).

Repeated capacity scaling. The repeated capacity scaling algorithm uses the same approach as the capacity scaling algorithm [GTT90]. The algorithm conducts a series of Δ capacity scaling phases starting from a Δ equal to $max(u_{ij})$. Unlike capacity scaling, which refines Δ until it equals one, the repeated capacity scaling algorithm takes advantage of one key network property: the network is guaranteed to contain an arc (i, j) with a flow $f_{ij} > 4N\Delta$ after at most $log(6N^2)$ Δ scaling phases. This property guarantees that for some optimal flow f, arc's (i, j) flow is positive (proof in [AMO93, §10.6]). The algorithm ceases refining Δ when it finds such an arc (i, j). Instead, it merges nodes i and j into a single new node k, and connects node k to all the nodes that previously were connected to either node i or j. The algorithm sets the capacity and reduced cost of the new arcs to the values their corresponding arcs had. Following, the repeated capacity scaling algorithm updates Δ to the current maximum arc capacity and resumes the Δ capacity scaling phases. The algorithm completes with an optimal flow after it transforms the pseudoflow into a feasible flow in one of the capacity scaling phases or, after it merges the network into a single node p with $b(p) = 0$.

Enhanced capacity scaling. Like repeated capacity scaling, the enhanced capacity scaling algorithm identifies arcs that are guaranteed to route flow in an optimal solution [Orl93]. By contrast, the algorithm does not merge nodes and does not start afresh with a new Δ after each merge. The algorithm takes advantage of another key network property: an arc (i, j) is guaranteed to have a positive flow in the remaining Δ-scaling phases, if it has $f_{ij} \geq 8N\Delta$ (proof in [AMO93, §10.7]). Arcs that respect this inequality are called abundant arcs. In each Δ-scaling phase, the algorithm discovers subgraphs consisting only of abundant arcs. Following, it pushes flow such that there is at most one node i that has flow supply $b(i) \neq 0$ in each such subgraph. Next, the algorithm conducts the same steps as capacity scaling to finish the Δ-scaling phase: it augments flow from supply nodes to sink nodes along paths on which it can push Δ units of flow. The algorithm completes with an optimal solution when there are no supply or sink nodes left in the the netowrk (i.e., the mass balance constraints are satisfied). The flow satisfies the reduced cost optimality conditions in each phase.

Time complexities. In Table 5.1, I summarise the worst-case complexities of the algorithms I discussed. The complexities suggest that successive shortest path ought to offer competitive runtimes. Its worst-case complexity is better than the complexity of: cycle canceling, minimum mean cycle canceling, primal-dual, out-of-kilter and repeated capacity scaling. Moreover, the algorithm should finish faster than capacity scaling and double scaling if $NU\log(N) < M\log(U)\log(N)$ and faster than enhanced capacity scaling if $N^2U < M^2$. However, since min-

Algorithm	Worst-case complexity
Cycle canceling	$O(NM^2CU)$
Minimum mean cycle canceling	$O(N^2M^2\log(NC))$
Successive shortest path	$O(N^2U\log(N))$
Primal-dual	$O(NU(M+N\log(N)+NM\log(N)))$
Out-of-kilter	$O(M^2U+MNU\log(N)))$
Relaxation	$O(M^3CU^2)$
Capacity scaling	$O((M^2+MN\log(N))\log(U))$
Cost scaling	$O(N^2M\log(NC))$
Double scaling	$O(NM\log(U)\log(NC))$
Repeated capacity scaling	$O(M^2\log(N)(M+N\log(N)))$
Enhanced capacity scaling	$O(M\log(N)(M+\log(N)))$

Table 5.1: Worst-case time complexities for the min-cost flow algorithms I describe. N is the number of nodes, M the number of arcs, C the largest arc cost and U the largest arc capacity. In the flow networks generated by scheduling policies, $M > N > C > U$. In bold I highlight the algorithms I implemented in Flowlessly.

cost flow algorithms are known to exhibit vastly different runtimes depending on the type of input graphs [Löb96; FM06; KK12], I decided to directly measure their performance.

Experiments. I implemented four min-cost flow algorithms in my Flowlessly solver: *(i)* the cycle canceling algorithm that is representative for the algorithms that maintain a feasible flow and iteratively refine the flow until it is optimal, *(ii)* the successive shortest path algorithm that has a competitive worst-case complexity, *(iii)* the relaxation algorithm that optimises the dual problem, and *(iv)* the cost scaling algorithm that works very well in practice [KK12]. I did not implement the other algorithms I discuss above because they either have worse worst-case complexity than one or more of the algorithms I implemented (e.g., minimum mean cycle canceling, primal-dual, out-of-kilter), or because they are scaling algorithms that do not work as well as cost scaling does in practice (e.g., capacity scaling, double scaling, repeated capacity scaling, enhanced capacity scaling). The algorithms I implemented are not strongly polynomial-time algorithms, but this does not negatively affect runtime because the flow networks scheduling policies generate always respect the inequality $M > N > C > U$ (i.e., they have more arcs than nodes, the number of nodes if greater than the maximum arc cost, which is higher than the maximum arc capacity).

In the experiment, I subsample the Google trace and replay it for simulated clusters of different sizes (like in Figure 5.5). I use the Quincy scheduling policy for batch jobs, and prioritise service jobs over batch ones. In Figure 5.8, I plot the average runtime for each min-cost algorithm I consider. Despite having the best worst-case time complexity, successive shortest path outperforms only cycle canceling, and even on a relatively small cluster of 1,250 machines its algorithm runtime exceeds 100 seconds.

Moreover, the relaxation algorithm, which has the highest worst-case time complexity out of the

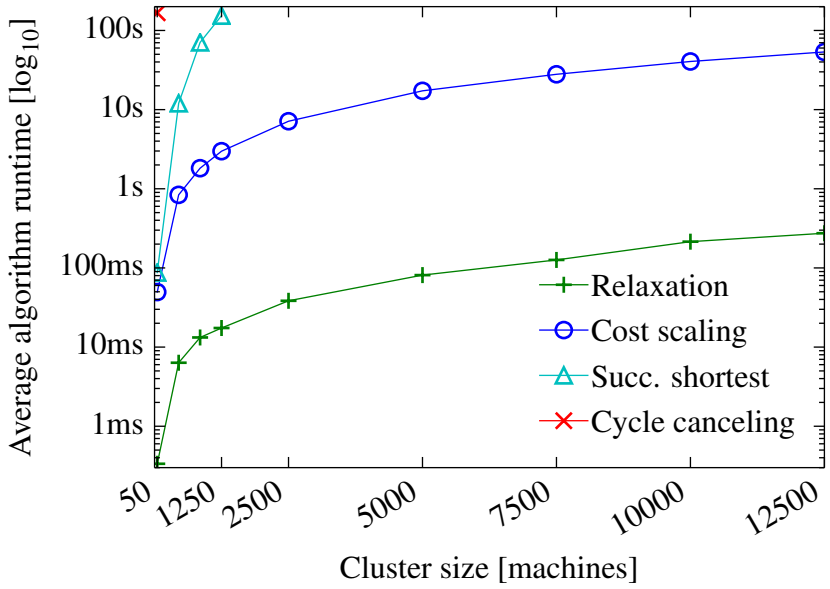


Figure 5.8: Average runtime for min-cost flow algorithms on clusters of different sizes, sub-sampled from the Google trace. I use the Quincy scheduling policy, and slot utilisation is about 50%. The relaxation algorithm performs best, despite having the highest time complexity.

algorithms I consider, performs best in practice. It outperforms cost scaling, which Quincy's solver uses, by two orders of magnitude. On average, it makes placements in under 200ms even on a full-size cluster of 12,500 machines. One key reason for this perhaps surprising performance is that the relaxation algorithm does minimal work when most scheduling choices are straightforward. This is the case when the destinations for tasks' flow are uncontested, i.e., not many new tasks have arcs to the same location and attempt to schedule there. In this situation, relaxation manages to route most of the flow supply in almost a single pass over the flow network.

5.3.2 Edge cases for relaxation

Relaxation is fast in the setup I described above, but there are cluster setups when relaxation is slow. For example, it can perform poorly on flow networks generated for highly loaded or oversubscribed clusters, situations that are common in batch processing clusters [BEL+14; RKK+16]. In Figure 5.9, I illustrate this: I push the simulated Google cluster closer to oversubscription. I take a snapshot of the cluster at 90% slot utilisation (i.e., 90% of the task slots already run tasks) and I submit increasingly larger jobs until all cluster slots are utilised and some tasks have to queue to be executed. Like in my previous experiments, I use the Quincy scheduling policy. The relaxation runtime increases rapidly, and at approximately 93% cluster slot utilisation, it exceeds that of cost scaling (the second best algorithm), growing to over 400s in the oversubscribed case.

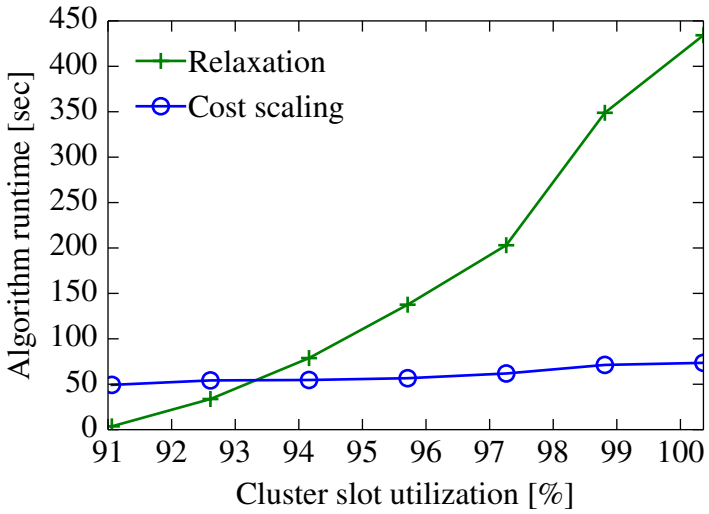


Figure 5.9: Close to full cluster slot utilisation, relaxation runtime increases dramatically, while cost scaling is unaffected: the *x*-axis shows the utilisation after scheduling jobs of increasing size to a 90%-utilised cluster.

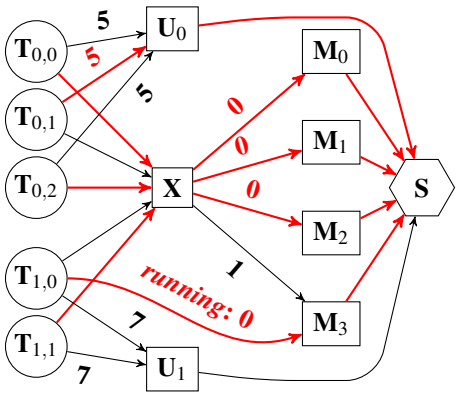


Figure 5.10: Load-spreading policy with single cluster aggregator (**X**) and costs proportional to number of tasks per machine.

Additionally, some scheduling policies generate challenging flow networks that *inherently* create contention among tasks, and thus chanllenge the relaxation algorithm. Consider, for example, a simple load-spreading policy that balances the number of tasks on each cluster machine. In Figure 5.10, I show an example flow network generated by this policy for a small cluster with four machines and five tasks. All task nodes only have arcs to a cluster-wide aggregator node (**X**) and to unscheduled aggregator nodes ($\mathbf{U}_j$). The cluster aggregator **X** connects with arcs to each machine node and the cost on each outgoing arc to each machine is proportional to the number of tasks already running on the machine (e.g., one task on $\mathbf{M}_3$). The effect is that the number of tasks on a machine only increases once all other machines have at least as many tasks. This policy makes "under-populated" machines a popular destination for tasks' flow, and thus creates contention.

I illustrate the effect the task contention created by the load-spreading policy has on min-cost

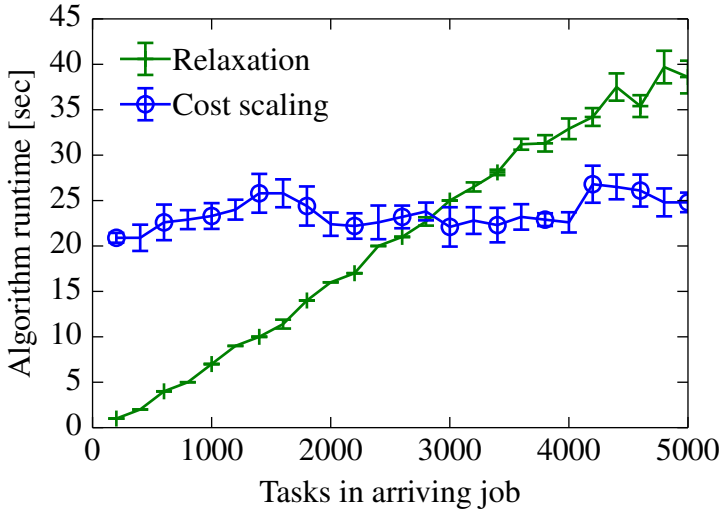


Figure 5.11: Contention slows down the relaxation algorithm: on cluster with a load-spreading scheduling policy, relaxation runtime exceeds that of cost scaling at just under 3,000 concurrently arriving tasks.

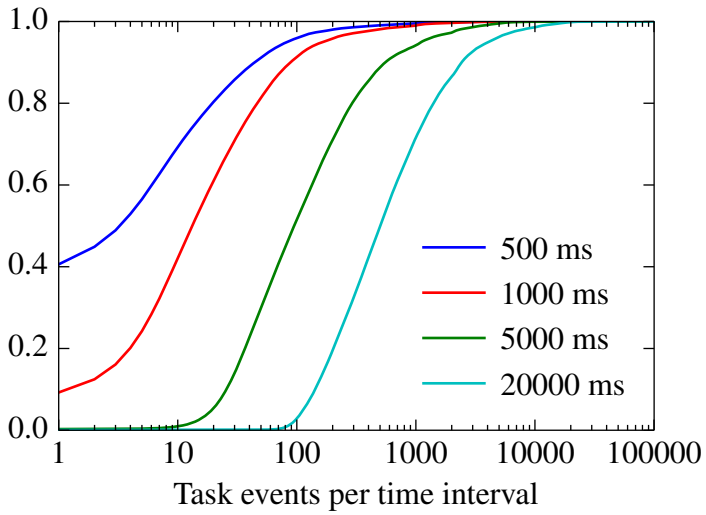


Figure 5.12: CDF of the number of scheduling events per various time intervals in the Google trace. The faster a scheduler runs the fewer tasks it must place.

flow algorithms runtime with an experiment. I submit a single job with an increasing number of tasks and measure how long it takes each algorithm to compute the optimal flow. This experiment simulates the rare-but-important arrival of very large jobs: for example, 1.2% of jobs in the Google trace have over 1,000 tasks, and some even over 20,000. Figure 5.11 shows that relaxation's runtime increases linearly in the number of tasks. Relaxation exceeds cost scaling's runtime when faced with over 3,000 new tasks.

To make matters worse, a single overlong relaxation run can have a devastating effect on long-term task placement latency. While relaxation runs, many tasks can finish and free slots. But these slots will not be used until the next relaxation run completes because the current flow

networkm on which relaxation computes the flow, does not encode the slots as available. Thus, Firmament artificially creates flow networks with more task contention compared to if it were to quickly utilise recently freed slots. Moreover, since many tasks may be submitted during such a long run, the scheduler may be again faced with a large number of new tasks when it runs again.

I analyse the workload in the public Google cluster trace to discover how many tasks the scheduler must handle depending on how fast it runs. I divide the 30-day trace into time windows of different sizes and count the number of task events (e.g., task submissions, failures, completions etc.) that occur in each time window. In Figure 5.12, I show CDFs of event count per time window for window sizes between 0.5s and 20s. A scheduler that completes a min-cost flow optimisation every 0.5s must process fewer than ten events in over 60% of cases, and fewer than 100 events in 95% of cases. If the scheduler runs every 20s, however, it must process over 500 task events in the median, and about 5,000 events in the 95th percentile. This highlights a vicious cycle: the quicker the scheduler completes, the less work it has to do in each scheduler run. Few long relaxation runs may have a disastrous effect: many task events accumulate while the algorithm runs, next run may take even longer, and thus the scheduler may even fail to ever recover to low task placement latencies. Thus, in the following sections I describe several techniques I developed to address these rare-but-important situations.

5.3.3 Approximate min-cost flow

Min-cost flow algorithms return an *optimal* solution. For the cluster scheduling problem, however, an approximate solution may well suffice. For example, TetriSched [TZP+16] (based on a mixed-integer linear programming solver), as well as Paragon [DK13] and Quasar [DK14] (based on collaborative filtering), terminate their solution search after a bounded amount of time. In this section, I investigate how good the solutions are if I terminate cost scaling and relaxation early. My hypothesis is that the algorithms may spend a long time on minor refinements to the solution with little impact on the overall task placements.

I run an experiment in which I use a highly-utilised cluster (same setup as in Figure 5.9) to investigate relaxation and cost scaling, but the results generalise other setups as well. In Figure 5.13, I show the number of "misplaced" tasks as a function of how early I terminate the algorithm. I treat any task as misplaced if it is *(i)* preempted in the approximate solution but keeps running in the optimal one; *(ii)* scheduled on a different machine to where it is scheduled in the optimal solution; *(iii)* left unscheduled but placed on a machine in the optimal solution. Both relaxation and cost scaling have thousands of misplaced tasks when terminated early, even on the final algorithm internal iteration before completing at 45s (cost scaling) and 142s (relaxation). The others algorithms I implemented in Flowlessly would misplace even more tasks when terminated early. Cycle canceling spends most of its runtime executing a max-flow algorithm to compute a feasible flow. If the algorithm is terminated early then it outputs a pseudoflow which is guaranteed to misplace at least as many tasks as much flow supply it has not yet

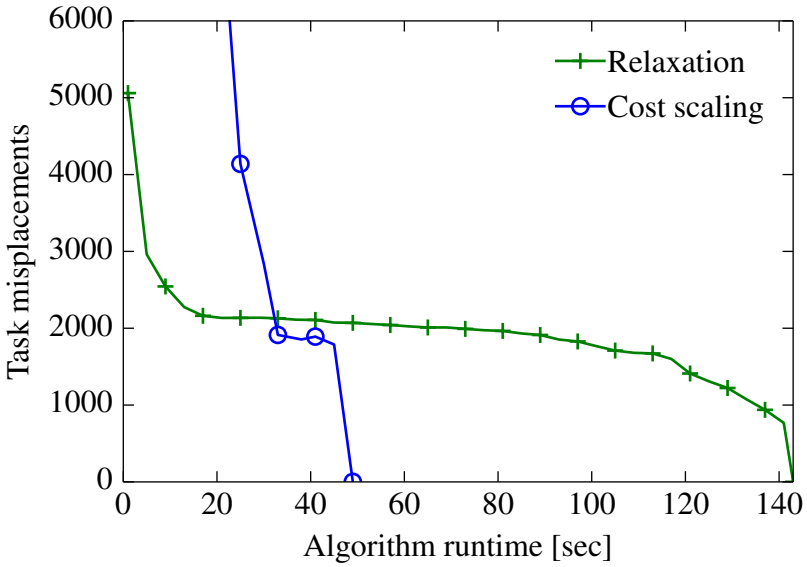


Figure 5.13: Approximate min-cost flow yields poor solutions, since many tasks are misplaced until shortly before the algorithms reach the optimal solution. The simulated cluster is at about 96% slot utilisation.

routed. Similarly, early termination of successive shortest path algorithm produces many task misplacements because the algorithm gradually reduces a pseudoflow's infeasibility by routing flow from source nodes to sink nodes. I thus conclude that early termination appears not to be a viable scalability optimisation for flow-based schedulers.

5.3.4 Incremental min-cost flow algorithms

One of my key insights is: the cluster state does not change dramatically between subsequent scheduling runs even when scheduling tasks on a large cluster. In a typical Google cluster, fewer than 100 task events happen in 95% of 500ms long time intervals (see Figure 5.12). Nonetheless, min-cost flow algorithms use a sledgehammer approach: they run from scratch over the entire flow network regardless of how many cluster events, which lead to flow network changes, occur between runs.

Min-cost flow algorithms might complete faster if they can reuse existing graph state and incrementally adjust the flow computed on the previous run. In this subsection, I describe the changes that are required to incrementalise min-cost flow algorithms, and provide some intuition for which algorithms are suitable for incremental use. I adjust Firmament to work incrementally. It collects scheduling events (e.g., task submissions, machine failures) while Flowlessly runs, updates its internal flow network and submits only graph changes to Flowlessly. Moreover, I change Flowlessly's algorithms to apply the graph changes received from Firmament on the latest flow solution, and to incrementally compute the new optimal flow.

All cluster events (e.g., task submissions, machine failures) ultimately reduce to three different types of *graph change* in the flow network:

1. **Supply changes** at nodes when arcs or nodes that previously carried flow are removed (e.g., due to machine failure), or when nodes with supply are added (e.g., at task submission).

2. **Capacity changes** on arcs due to machines failing or joining the cluster. Note that arc additions and removals can also be modelled as capacity changes from and to zero capacity arcs (i.e., new tasks preferences are modelled as capacity changes).

3. **Cost changes** on arcs when the desirability of routing flow via some arcs changes; when these happen exactly depends on the scheduling policy (e.g., load on a resource has changed, a task has waited for a long time to run).

Changes that adjust a node's supply, an arc's capacity or cost can invalidate flow feasibility and optimality. However, as I noted in §5.3.1, many min-cost flow algorithms require the flow to be either optimal or feasible before each internal iteration. In Table 5.2, I summarize these requirements. Some algorithms (e.g., cycle canceling, cost scaling) require the flow to be feasible at each step and work towards achieving one of the equivalent types of optimality: negative cycle optimality, reduced cost optimality or complementary slackness optimality. Other algorithms (e.g., successive shortest path, relaxation) require the flow to be optimal, but not necessarily feasible before each step. These algorithms push flow supply to the sink in order for more nodes to satisfy the mass balance constraints, adjust flow on arcs such that more arcs satisfy the capacity constraints, and ultimately find a feasible flow. The incremental solution requires the flow to be both optimal and feasible because an infeasible flow would evict tasks or leave tasks unscheduled, and a non-optimal solution would misplace tasks.

I handle incremental graph changes differently depending on the preconditions the algorithm must satisfy before each internal iteration. I now describe how I handle graph changes in the four possible scenarios:

- For the algorithms that require the **mass balance constraints** to be satisfied, I route flow supply resulted from *supply changes* (e.g., node addition, node removal) to sink nodes. I use a max-flow algorithm if the graphs has many supply nodes. Otherwise, I iteratively push flow along the shortest path route. *Arc cost* and *capacity changes* do not break the mass balance constraints, unless arcs are removed. When handling an arc removal, I first drain the arc's flow, and thus I can treat it as two changes: *(i)* an increase in the arc's source node supply, and *(ii)* a decrease in the arc's destination node supply.

- For algorithms that require the flow to be **feasible** (i.e., satisfy capacity and mass balance constraints), I use max-flow or shortest path algorithms to obtain a flow that satisfies again the mass balance constraints after I apply the *supply changes*. But before I apply these algorithms, I ensure the flow satisfies the capacity constraints. Two types of *arc changes* can break capacity constraints: *(i)* changes that decrease arc capacities, and *(ii)* changes that increase minimum arc flow requirements – I treat additions of arcs with minimum

Algorithm	*Feasibility*		*Optimality*	
	Capacity cnstr.	*Mass balance cnstr.*	*Reduced cost*	*ε-optimality*
Cycle canceling	✓	✓	–	–
Min mean cycle canceling	✓	✓	–	–
Successive shortest path	✓	–	✓	–
Primal-dual	✓	–	✓	–
Out-of-kilter	–	✓	–	–
Relaxation	✓	–	✓	–
Capacity scaling	✓	–	✓	–
Cost scaling	✓	(✓)†	–	✓
Double scaling	✓	✓	–	✓
Repeated capacity scaling	✓	–	✓	–
Enhanced capacity scaling	✓	–	✓	–

Table 5.2: Algorithms – the ones implemented in Flowlessly are in **bold** – have different preconditions for each internal iteration. Some algorithms expect both types of feasibility constraints to be satisfied, while others only require reduced cost optimality constraints to be satisfied. Double scaling expects capacity constraints, mass balance constraints and ε-optimality, making it difficult to incrementalise. Cost scaling requires flow to satisfy mass balance constrains (†), but a modified version that does not have this requirement, but has a worse worst-case complexity exists. I incrementalised this cost scaling version in Flowlessly.

flow requirement as increases in their minimum flow. For both change types, I adjust the flow on the changed arc such capacity constraints are satisfied, and as a result, I cause a change in the arc's source and destionation nodes supply.

- Some algorithms (e.g., successive shortest path, primal-dial) require the flow to satisfy both **capacity constraints** and the **reduced cost optimality conditions** before each internal iteration. For these algorithms, I first apply *capacity changes* and adjust the flow such that it continues to satisfy the capacity constraints. Moreover, both *capacity changes* and arc *cost changes* can introduce arcs that do not respect the reduced cost optimality conditions (i.e., residual arcs with negative reduced cost). After I apply these changes, I push flow on the affected arcs up to capacity, and thus I re-establish reduced cost optimality. For these algorithms, I do not take any actions on *node changes* because these changes do not break capacity constraints and reduced cost optimality conditions.

- Some algorithms (e.g., double scaling) require the flow to satisfy **mass balance constraints**, **capacity constraints** and **ε-optimality**. For these algorithms, I first apply graph changes and adjust flow on any arcs that do not satisfy the capacity constraints. Subsequently, I run a max-flow algorithm or a successive shortest path algorithm to obtain a feasible flow. Finally, I compute the ε-optimality of the current flow and resume the min-cost flow algorithm with the new ε value.

I considered implementing an incremental version of each algorithm from Table 5.2. However, some algorithms are unsuitable to incrementally re-compute the optimal flow. For example, the

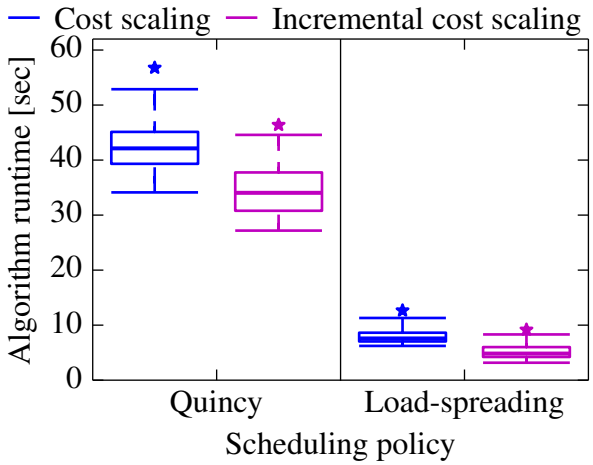


Figure 5.14: Incremental cost scaling is 25% faster compared to from-scratch cost scaling for the Quincy policy and 50% faster for the load-spreading policy.

min-cost flow algorithms that require the mass balance constraints to be satisfied before each internal iteration are unlikely to show significant runtime reductions. In order for the flow to satisfy again the mass balance constraints, I have to use max-flow or shortest path algorithms to adjust it. However, these algorithms are expensive to execute; often their average-case performance is close to their worst-case – $O(FM)$ worst-case complexity for the flow networks generated by schedulers, where F is the total flow supply. Common cluster events (e.g., task submission, completion or failure) cause flow supply in the graph, and thus incremental algorithm runtime.

Capacity scaling and other algorithms similar algorithms (e.g., repeated capacity scaling, enhanced capacity scaling) are also unlikely to quickly re-compute the optimal flow. They do not require mass balance constraints to be satisfied, but they require the residual flow network to only contain nodes with supply greater than $-\Delta$ and smaller than Δ. Many common cluster events (e.g., task submission, task completion, machine utilisation change) introduce large node flow supplies. When such events occur, the incremental min-cost algorithms based on capacity scaling cannot quickly re-compute the solution because they fallback to Δ values almost as large as if the algorithms were to start them from scratch.

I implemented incremental versions of the cycle canceling, successive shortest path, cost scaling and relaxation algorithms. However, I only discuss the algorithms that have competitive runtimes. **Incremental cost scaling** is up to 50% faster than running cost scaling from scratch (Figure 5.14). Cost scaling's possible gains from incremental execution are limited, because cost scaling requires the flow to be feasible and ε-optimal before each intermediate iteration (Table 5.2).

Graph changes can cause the flow to violate one or both requirements. Table 5.3 shows the effect of different types of changes with respect of cost scaling's flow feasibility and optimality requirements. For example, a change that modifies the cost of an arc (i, j) from $c_{ij}^{\pi} < 0$

	Reduced cost on arc from i to j		
Change type	$c^\pi_{ij} < 0$	$c^\pi_{ij} = 0$	$c^\pi_{ij} > 0$
Increasing arc cap.			
Decreasing arc cap.		$f_{ij} > u\prime_{ij}$	
Increasing arc cost	$c\prime^\pi_{ij} > 0$	$f_{ij} > 0$	
Decreasing arc cost			$c\prime^\pi_{ij} < 0$

Table 5.3: Arc changes that require solution reoptimisation. *Green:* flow is still optimal and feasible; *red:* change breaks feasibility or optimality; *orange:* change breaks feasibility or optimality if condition in cell is true. Only decreasing arc capacity can break feasibility; the other changes affect optimality only.

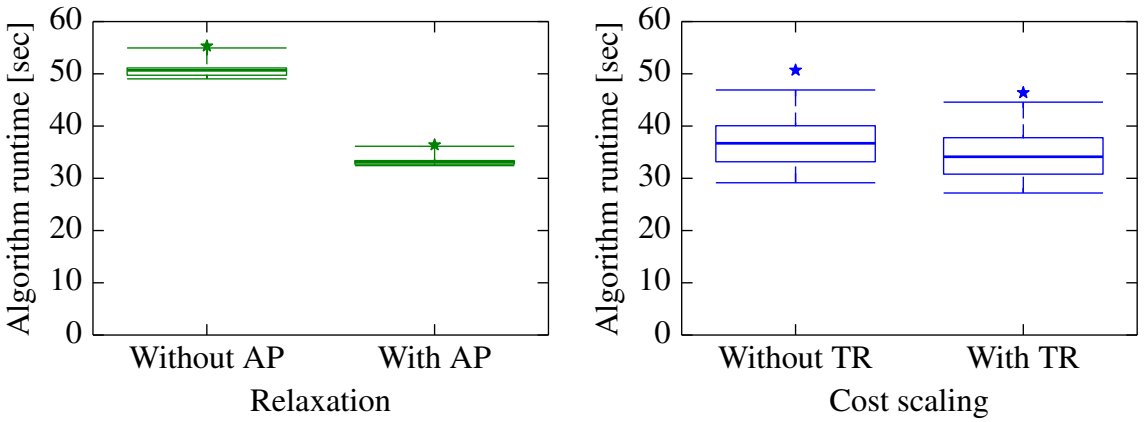
to $c\prime^\pi_{ij} > 0$ breaks optimality, whereas additions or removals of task nodes (with supply) break feasibility. My cost scaling implementation does not have the best possible worst-case complexity, but it does not require the mass balance constraints to be satisfied before each internal iteration. Nevertheless, my incremental cost scaling does not run several times faster than the from-scratch version. Many changes affect optimality and require cost scaling to fall back to a higher ε-optimality to compensate. In order to bring ε back down, my cost scaling implementation must do a substantial part of the work it would do from scratch. However, the limited improvement still helps reduce the runtime of the second best algorithm.

Incremental relaxation ought to work much better than incremental cost scaling because the relaxation algorithm only needs the flow to satisfy the capacity constraints and the reduced cost optimality conditions. However, in practice it turns out not to work well. While the algorithm can be incrementalised with relative ease, it can – counter-intuitively – be slower incrementally than when running from scratch.

Relaxation requires reduced cost optimality to hold at each step of the algorithm and improves feasibility (Table 5.2) by pushing flow on zero-reduced cost arcs from source nodes to sink nodes. The algorithm gradually constructs a tree of arcs with zero-reduce costs for each source node in order to find zero-reduce cost paths to sink nodes. Incremental relaxation works with the existing, close-to-optimal state, which increases runtime because the closer to optimality the solution is, the larger the trees to be built are. In contrast, when running from scratch, only a small number of source nodes (i.e., tasks) selected in the execution have large trees associated with them. In practice, I have found that incremental relaxation can perform well in cases when tasks are unconnected to zero-reduced cost tree. However, it can perform poorly, especially in challenging situations with many tasks, since these are likely to require many tree constructions.

5.3.5 Optimising min-cost flow algorithms

Relaxation has promising common-case performance at scale for typical workloads, but its edge-case behaviour makes it necessary to either *(i)* use other algorithms in these cases, or *(ii)* apply heuristics developed for these cases.



(a) *Arc prioritisation* (AP) reduces relaxation's runtime by 45%.

(b) *Efficient task removal* (TR) reduces incremental cost scaling's runtime by 10%.

Figure 5.15: Problem-specific heuristics reduce min-cost flow runtimes.

My scheduler runs min-cost flow algorithms on flow networks with specific properties, which differ from the flow networks typically used to evaluate min-cost flow algorithms [KK12, §4]. For example, the flow networks Firmament's scheduling policies generate have a single sink; are directed acyclic graphs, and flow must always traverse unscheduled aggregators or machine nodes. Hence, problem specific heuristics might help min-cost flow algorithms find solutions more quickly. I developed several such heuristics, and found two beneficial ones: *(i)* prioritisation of promising arcs, and *(ii)* efficient task node removal.

5.3.5.1 Arc prioritisation

The relaxation algorithm builds for every supply node a tree of zero-reduced cost arcs (see §5.3.4) in order to locate zero-reduced paths (i.e., paths that do not break reduced cost optimality) to nodes with demand. When the algorithm builds this tree, it can extend the tree with any zero-reduced cost arc that connects a node inside the tree to a node outside it. However, some arcs are better choices for tree extension than others because the quicker the algorithm can find paths to nodes with demand, the sooner it can route the supply. For example, Firmament's flow networks contain arcs that indicate a task's machine placement preference. These arcs connect tasks to resource nodes which in turn are few arcs away or directly connected to the sink node. I adjust relaxation to prioritise such arcs that lead to nodes with demand when it extends the zero-reduced cost tree. I add these arcs to the front of relaxation's priority queue to ensure they are visited sooner.

In effect, this heuristic implements a hybrid graph traversal that biases towards depth-first exploration when demand nodes can be reached, but breadth first exploration otherwise. In Figure 5.15a, I show that this heuristic reduces relaxation runtime by 45% when running over a flow network with contended nodes.

5.3.5.2 Efficient task removal

My second heuristic reduces incremental min-cost flow algorithms runtime. The key insight that lead the development of this heuristic is: removing a running task (e.g., due to its completion, preemption, or a machine failure) constitutes a common, but costly change. When Firmament removes a task node, it must also remove the arcs connected to the node. At least one of these arcs carried flow in the previous optimal solution. Thus, when Firmament removes this arc, it breaks feasibility and creates demand at the other node the arc is connected to.

I improve task removal by adding an arc, called *running arc*, from each node representing a running task to the node modelling the machine on which this task runs. I set the cost on this arc to be equal to the cost of continuing to run the task on the machine its currently running on. Subsequent min-cost flow scheduling runs route the task's flow supply along this arc unless better alternatives become available. Given that the task's flow is pushed along the running arc, I can easily reconstruct the task's flow supply path through the graph and remove the flow all the way to the single sink node. This creates demand in only a single place (the sink), which accelerates the incremental solution, as fewer node potentials need to be adjusted. In Figure 5.15b, I show that this heuristic reduces incremental cost scaling runtime by about 10%.[2]

5.3.6 Algorithm choice

In §5.3.1 and §5.3.2, I show that min-cost flow algorithm real-world performance varies and that no algorithm consistently outperforms all others. Relaxation often works best in practice, but scales poorly in cases when resources are heavily contended. Cost scaling, by contrast, scales well and can be incrementalised (§5.3.4), but is usually substantially slower than relaxation. Not even the heuristic I previously described do not reduce runtime such that a single algorithm offers low task placement latency for all scheduling policies and cluster scenarios.

Flowlessly always speculatively executes cost scaling and relaxation, and picks the solution offered by whichever algorithm finishes first. In the common case, this is relaxation; however, in challenging situations when relaxation is slow (e.g., many contended resources), cost scaling guarantees that Firmament's placement latency does not grow unreasonably large. I run both algorithms instead of developing a heuristic to choose the right one for two reasons. First, it is cheap; the algorithms are single-threaded and do not parallelise well. They comprise of many steps that make small changes to data that would have to be shared among threads, and thus the parallel implementations would have to use locks extensively. I parallelised the cost scaling algorithm, but I found my implementation to barely outperform the single-threaded algorithm in the common case, and to take longer to complete on some flow networks. The second reason for which I run both algorithm is that predicting the right algorithm is hard and the heuristics would depend on both scheduling policy and cluster utilisation, making it brittle and complex.

[2]Relaxation does not benefit from this heuristic because the algorithm is not suitable for running incrementally (§5.3.4).

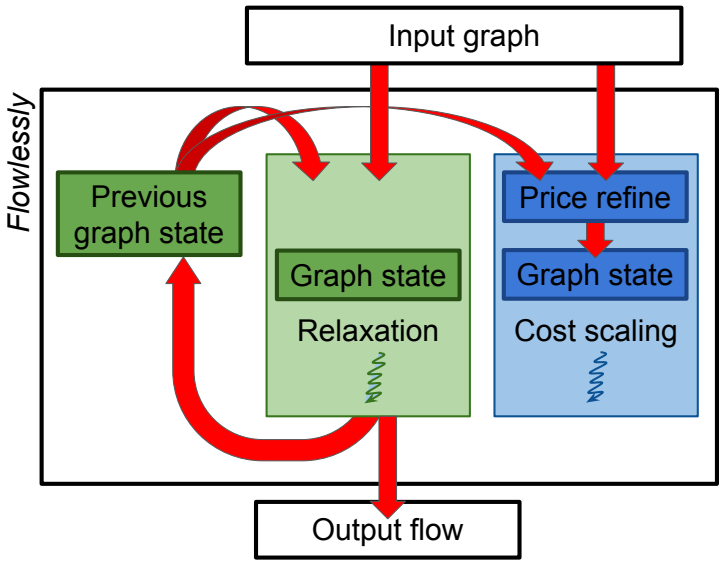


Figure 5.16: Schematic of Flowlessly's internals.

In Figure 5.16, I show a high-level view of how Flowlessly runs two algorithms in parallel. In order to efficiently transition state from the relaxation algorithm to incremental cost scaling, Flowlessly applies an optimisation that I describe now.

5.3.6.1 Efficient algorithm switching

Flowlessly runs in parallel relaxation and incremental cost scaling – because it is substantially faster than non-incremental cost scaling (§5.3.4). In the common case, however, the (from-scratch) relaxation algorithm is first to finish. Therefore, the next incremental cost scaling run must use the optimised solution from *relaxation* as a starting point. This can be unnecessarily slow because relaxation and cost scaling maintain different reduced cost graph representations. Specifically, relaxation can converge on node potentials that are poor choices for satisfying cost scaling's complementary slackness optimality requirements because relaxation adjusts node potentials and flow to satisfy reduced cost optimality conditions. For example, relaxation can converge on greatly different potentials on two non-adjacent nodes i and j. But graph changes generated by cluster events can introduce arc (i, j), which has a high reduced cost c_{ij}. This type of graph changes are difficult to handle because incremental cost scaling typically requires the flow to both satisfy mass balance constraints and to be ε-optimal (for as smallest possible ε). The incremental algorithm can either: *(i)* exhaust the arc's residual capacity, but break mass balance constraints, or *(ii)* leave flow unchanged, but deteriorate ε-optimality to a bigger ε. In Flowlessly, I choose the later approach and instead use a heuristic to reduce differences between adjacent node potentials and improve ε-optimality.

I found that a heuristic originally developed for use within cost scaling, *price refine* [Gol97], helps make the transition between relaxation and incremental cost scaling more efficient. Price refine adjusts node potentials to discover if an ε-optimal flow is ε/α-optimal as well. I apply

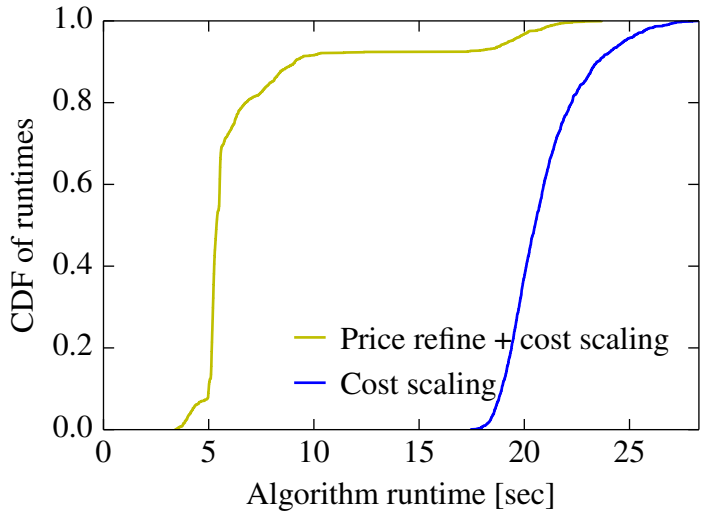


Figure 5.17: Applying the price refine heuristic to a graph coming from relaxation reduces incremental cost scaling runtime by 4×.

price refine on the previous solution *before* I apply the latest cluster changes. The flow returned by the previous algorithm run is $\frac{1}{n}$-optimal. Thus, I can reset all node potentials to zero and invoke price refine to compute node potentials for the $\frac{1}{n}$-optimal flow. Given the flow optimality, price refine is guaranteed to find without modifying the flow changes, node potentials that satisfy complementary slackness optimality conditions. Hence, incremental cost scaling only has to restart at an ε value equal to the maximum arc cost graph change.

In Figure 5.17, I illustrate that my price refine implementation reduces incremental cost scaling runtime by about 4× in 90% of the cases when I apply it on relaxantion's graph output.

5.3.7 Efficient solver interaction

So far, I focused on reducing Flowlessly's algorithm runtime. However, I must do more to achieve low task placement latencies. First, Firmament must efficiently update flow network nodes, arcs, costs, and capacities before every min-cost flow optimisation to reflect the chosen scheduling policy. Second, Firmament must not generate superfluous incremental flow network changes. Third, Firmament must quickly extract task placements out of the flow network after each optimisation finishes. All of these steps are not included in Flowlessly runtime, but must be efficient for placement latency to be low. I improve over the prior work on min-cost flow-based scheduling in Quincy for these aspects, as I explain next.

Flow network updates. I optimise Firmament to run only two bread-first traversals (BFS) over the flow network to update the network before another solver run. The first traversal updates statistics associated with tasks and resources (e.g., current machine utilization for machine nodes, resource utilization and requests for task nodes). The traversal starts from nodes adjacent

to the sink (usually machine nodes), and propagates statistics backwards up to task nodes along each node's incoming arcs. When the traversal completes, each node has up-to-date resources metadata statistics, and if required, statistics about the descendant resources. By contrast, the second traversal starts at task nodes. For each node, it invokes scheduling policy API methods that are implemented by every supported scheduling policy [Sch16, Appendix C]. These methods add and remove flow network nodes, add and remove arcs, and change arc costs and capacities using the metadata statistics computed by the first traversal. For example, the methods can remove task preference arcs to machines, if the scheduler recently placed tasks which increased utlisation on these machines. The two traversals Flowlessly runs to update the flow network have a linear in the number of arcs worst-case complexity of $O(M+N)$. Their runtime is negligible compared to min-cost flow algorithms runtime, which have higher worst-case complexities (see Table 5.1).

Flow network changes creation. Min-cost flow solvers receive entire flow networks as input and output optimal flow. In each scheduling round they start from scratch. By contrast, Flowlessly stores the flow network across scheduling rounds, expects to receive only flow network changes, and continously adjusts the flow. I extended Firmament to only submit flow network changes to Flowlessly before each solver run. Nonetheless, these flow network changes can be expensive to process because they may require Flowlessly to resize its internal flow network data structures, or may cause numerous cache misses. I reduce how many flow network changes Firmament generates by making sure it:

- does not generate duplicate changes;
- merges multiple changes to an arc into a single change that encompasses all;
- prunes out changes to incoming and outgoing arcs of nodes that it later removes.

My optimisations reduce how long it takes Flowlessly to modify the flow network between consecutive incremental min-cost flow runs and, ultimately, improve task placement latency.

Task placement extraction. Flowlessly returns an optimal flow at the end of each run out of which Firmament extracts task placements. The Quincy scheduler extracts task placements for its policy, but I had to generalise its approach because Firmament supports minimum flow requirements on arcs, arbitrary aggregators and multiple arcs between two nodes in the flow network. There may be many optimal flow paths between task and machine pairs in the flow networks Firmament generates. These paths may be longer than in Quincy, where arcs necessarily pointed to machines or racks. I devised a graph traversal algorithm (see Listing 5.1) to extract task assignments efficiently. The algorithm starts from machine nodes and backwards traverses the graph using incoming arcs, propagating a list of machines to which each node sent flow. When the algorithm completes, a single-machine list has propagated to each scheduled

```
to_visit = machine_nodes  # list of machine nodes
node_flow_destinations = {}  # auxiliary remember set
mappings = {}  # final task mappings
while not to_visit.empty():
  node = to_visit.pop()
  if node.type is not TASK_NODE:
    # Visit the incoming arcs
    for arc in node.incoming_arcs():
      moved_machines = 0
      # Move as many machines to the incoming arc's
      # source node as there is flow on the arc
      while assigned_machines < arc.flow:
        node_flow_destinations[arc.source].append(
          node_flow_destinations[node].pop())
        moved_machines += 1
      # (Re)visit the incoming arc's source node
      if arc.source not in to_visit:
        to_visit.append(arc.source)
  else:  # node.type is TASK_NODE
    mappings[node.task_id] =
      node_flow_destinations[node].pop()
return mappings
```

Listing 5.1: Algorithm for extracting task placements from the flow returned by the solver.

task node. In contrast to standard breadth-first and depth-first graph traversals, the algorithm may visit a node more than once if there are several flow paths of different length between the sink and the node. However, such situations are rare in the flow networks Firmament's scheduling policies generate. In the common case, the algorithm extracts the task placements in a single pass over the graph.

5.4 Extensions to min-cost flow-based scheduling

Prior work shows that min-cost flow-based schedulers choose task placements that offer good data locality [IPC+09], avoid task co-location interference [Sch16, §5.5.3], leverage hardware heterogeneity to reduce power consumption [Sch16, §5.5.4], support soft and hard constraints [Sch16, §5.4.1], and can offer fair shares of computers to jobs [IPC+09]. However, these min-cost flow-based schedulers still have several limitations:

Linear task placement costs: min-cost flow-based schedulers often simultaneously place several tasks on a resource. The cost of placing a task on a resource is given by the sum of the cost of the arcs along flow path from the task node to the resource node. These arc costs are statically computed by the scheduling policy before each min-cost flow solver runs. Thus, min-cost flow-based schedulers cannot account for the performance penalties two or more *simultaneously placed* tasks incur due to co-location (see Figure 5.18).

This limitation causes task makespan to increase or task application level performance to decrease. Quincy adopts a radical approach to solve this limitation: it only executes one task per machine. However, this approach leads to low cluster resource utilisation. Schwarzkopf suggests an alternative approach in which several tasks can be co-located, but schedulers cannot simultaneously place two or more tasks on a resource in a scheduling round [Sch16, §5.5.3]. This approach does not significantly decrease resource utilisation, but increases task placement latency because tasks may have to wait for several scheduling rounds before they are placed.

Tasks cannot have complex constraints: some cluster workloads have complex constraints that have mutual dependencies between tasks. Two popular types of such constraints are task affinity and task anti-affinity (§2.3.1.5). Task affinity constrains create dependencies between two or more tasks for being placed on the same resource (e.g., a web server task must share a machine with a database task). By contrast, task anti-affinity constraints restrict tasks to only be placed on different resources (e.g., distribute database tasks across machines to improve fault tolerance). Despite the appeal, there are no ways to specify such constraints in min-cost flow-based schedulers.

Inefficient gang scheduling: some cluster workloads expect schedulers to simultaneously place tasks on machine. For example, graph processing workflows are iterative and run in systems that require all tasks to synchronise at the end of each iteration (§ 2.1). Task waste resources and workflow makespan increases, if they are not placed simultaneously (i.e., gang scheduled). However, min-cost flow-based schedulers either do not support gang scheduling [IPC+09], or can only gang schedule a workflow at a time [Sch16, §5.4.3].

Support only job-based fairness: in order to choose high-quality placements and efficiently utilise clusters, schedulers must achieve good bin-packing on different resource dimensions (§ 2.3.1.3) and fairly share these different resources (§ 2.3.1.7). The Quincy scheduler explores trades-offs of task competing demands of data locality, fairness, and starvation-freedom. However, Quincy only fairly shares disk bandwidth among jobs at the machine granularity level. Schwarzkopf shows how min-cost flow schedulers can be extended to consider several resource dimensions, but his approach can only offer fair shares of running tasks or allocated machines to jobs [Sch16, §5.5.3]. Min-cost flow schedulers do not currently support other types of fairness (e.g., dominant resource fairness).

In this section, I introduce three new basic flow network concepts I developed: convex arc costs, "xor" flow network construct, and "and" flow network construct. I use these concepts as building blocks for building complex scheduling policies that address the limitations I described above. In Subsection 5.4.1, I explain how convex arcs costs can be used to avoid interfering task placements. Following, I describe how the basic flow network constructs can be grouped together to express complex constraints in flow networks, and to gang schedule many jobs at a

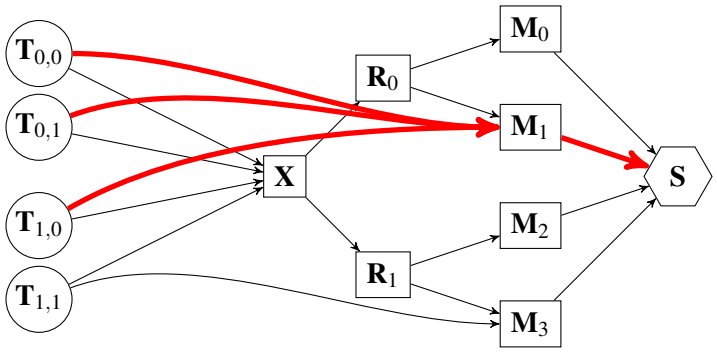


Figure 5.18: Example of min-cost flow scheduler's limitation in handling data skews. The scheduler places tasks $T_{0,0}$, $T_{0,1}$ and $T_{1,0}$ on machine M_1 without taking into account that the tasks may interfere.

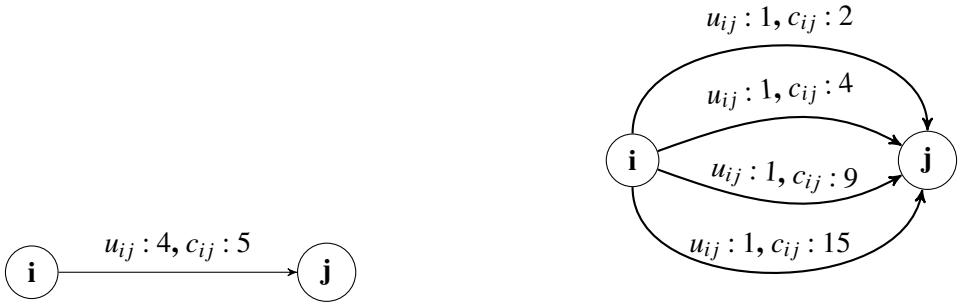
time (§ 5.4.2). Finally, I explain how min-cost flow-based schedulers can use these flow network construct to model different types of fairness in flow networks (§ 5.4.4).

5.4.1 Non-linear arc costs

Firmament can run scheduling policies that mitigate straggler tasks' effect on job completion time. These policies do not connect tasks to heavily loaded machines (i.e., blacklist machines) and connect task nodes with preference arcs to machine nodes on which task input data resides. Tasks placed by scheduling policies with preference arcs read locally more data, but the preference arcs can cause some tasks to become stragglers if: *(i)* data are singly replicated, or *(ii)* a fraction of data is used by many tasks [NEF+12]. These data distributions are not uncommon: 18% of the data from one of Microsoft's Bing Dryad clusters is accessed by at least three unique jobs at a time, and the top 12% most popular data are accessed over ten times more than the bottom third of the data [AAK+11],

In such situations, the Quincy scheduling policy creates many preferences arcs that point to several machines, which store the popular data. In Figure 5.18, I show one such example. Machine M_1 stores data that is accessed by tasks $T_{0,0}$, $T_{0,1}$ and $T_{1,0}$. They each have a preference arc to machine M_1. All min-cost flow solver executes, pushes flow along the preference arcs, and thus co-locate the three tasks on machine M_1. As a result, the scheduler may place too many interfering tasks on the machine in a scheduling round. This happens because preference arc costs are independent. In other words, the scheduler is just as likely to place an additional task on machine M_1 after it already decided to place several tasks in the current scheduling round.

Existing min-cost flow scheduler choose practical, but inefficient solutions to avoid this limitation. For example, the Quincy does not run more than a task per machine, which does not keep the clusters highly utilised. Schwarzkopf introduces an alternative approach based on admission control [Sch16, §5.5.3]. In his approach, the scheduler ensures that no more than a given number of tasks can be placed on a resource in a scheduling round. The scheduler sets an arc capacity equal to the maximum number of allowed co-located tasks on the arcs that connect



(a) Linear arc cost.

(b) Convex arc costs.

Figure 5.19: Example showing how convex arc costs can be modelled in the flow-network. An arc (i, j) is transformed into several arcs with different costs. The same amount of flow can be routed from node i to node j, but the cost increases linearly (5.19a) or non-linearly (5.19b).

resources nodes and the sink node. Schedulers that use Schwarzkopf's approach can achieve high resource utilisation, but they cannot offer low task placement latency for tasks that are admission controlled. These tasks wait for several scheduling rounds before they are placed. However, this trade-off is not necessary if scheduling policies use the flow network construct I introduce next.

Convex arc costs In all the flow networks I presented to this point, each arc (i, j) has associated a cost c_{ij} and a maximum flow capacity u_{ij}. In these flow networks, regardless of how many units of flows are already sent along arc (i, j), it costs c_{ij} to send an additional unit of flow (i.e., it costs $x * c_{ij}$ to send x units of flow). However, there are cases when it is desirable for the cost of sending flow to increase more than linearly. For example, if Firmament pushes two tasks' flow supply to a machine node along an arc, it may be desirable to cost more to push second task's flow supply because the task may interfere with the other task.

Firmament is not limited to use only arcs with linear costs in the flow networks. It can model *convex arc costs* by creating multiple arcs between pairs of nodes. These arcs can have different costs, minimum flow requirements and flow capacity constraints. For example, an arc (i, j) with a linear cost c_{ij} and a capacity u_{ij} can be transformed into several arcs of different cost and that connect node i and node j. The sum of capacities of these arcs must be equal to u_{ij}; same amount of flow can be routed along the multiple arcs that model arc (i, j), albeit at a cost that does not have to increase linearly. In Figure 5.19, I show how an arc with a flow capacity of 4 and a cost of 5, can be represented as several arcs that have an equal total capacity, but model a convex cost.

Scheduling policies that use convex arc costs generate flow networks with many arcs. These policies can introduce as many new arcs as the capacity of of prior linear cost arcs if they strive to achieve high arc cost accuracy. The runtime of existing min-cost flow solvers (e.g, cs2) increases on such large flow networks. By contrast, Flowlessly's runtime does not significantly increase when the flow networks contain more arcs (see Subsection 6.3.1).

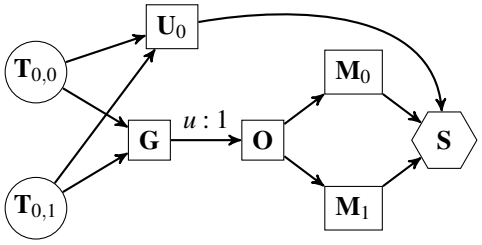
5.4.2 Complex placement constraints

Many tasks have placement constraints because not all cluster resources are equally suitable to execute them. In Section 2.3.1.5, I categorised placement contraints into three categories: *(i)* **soft constraints** that indicate placement preferences that not necessarily have to be satisfied, *(ii)* **hard constraints** express placement requirements that must be satisfied for individual tasks, and *(iii)* **complex constraints** that can be hard or soft, and require several tasks and machines to simultaneously satisfy them.

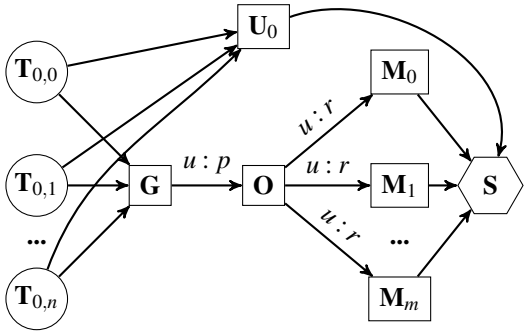
Placement constraints are common in practice: 50% of tasks from a Google cluster have soft or hard constraints related to machine properties, and 11% of tasks have complex constraints [SCH+11]. However, only few cluster schedulers support complex constraints [TCG+12; TZP+16]. In many cases, this lack of support is the result of a mismatch between the features schedulers need to support complex constraints and the architectures they use. Complex constraints create mutual dependencies between several tasks and machines. But in practice many schedulers cannot satisfy these dependencies because schedulers are queue-based and place tasks one by one (see § 5.1.1). Min-cost flow schedulers consider entire workloads in each scheduling round, albeit they do not support complex constraints [IPC+09; Sch16]. The flow networks these schedulers generate do not encode dependencies between tasks and machines, and thus solvers route tasks flow supply to sink nodes independently. To my knowledge, there is no min-cost flow cluster scheduler that offers complex constraints. Prior work claims that complex constraints cannot be expressed in flow networks [IPC+09; Sch16]. I show this is not the case: complex constraints can be represented in flow networks using two flow constructs that I introduce now.

"xor" flow network construct It is often desirable to model in the flow network exclusive disjunctions. These can be useful to express policies that avoid co-locating interfering tasks or that spread tasks across machines/racks for various reasons (e.g., to improve fault tolerance). In Figure 5.20a, I model the "xor" logical operation for two tasks' flow supply. I connect the tasks to an aggregator node (G) I introduce. I then connect this node only to another node aggregator (O). I set a maximum capacity of one on arc (G, O) to ensure that only one unit of flow can be routed along this arc. Thus, at most one task (i.e., $T_{0,0}$ or $T_{0,1}$) will be placed on the machines to which node O connects. In Figure 5.20b, I extend the construct to n tasks. Like previously, I connect task nodes to an aggregator node G that only has one outgoing arc to node O. I set the maximum capacity on this arc to the maximum number of tasks connected to node G that are allowed to be placed. I also set maximum capacities on the outgoing arcs from node O to machine nodes. These capacities control how many tasks are co-located on each machine.

The "xor" network flow construct can be used to express complex *conditional preemption constraints* such as if task T_x is placed, then task T_y must be preempted as a result. Similarly, the generalised "xor" construct can be used to express conditional constraints, but also complex *co-scheduling constraints* (such as task anti-affinity constraints) that require several tasks of a job to not share a resource (e.g., rack, machine).



(a) Example of "xor" flow network construct. Only one of T_0 and T_1 can be placed.



(b) Example of generalised "xor" flow network construct. No more than r tasks can be placed on a machine, and a maximum of p tasks out of n can be placed.

Figure 5.20: Examples that create dependencies between tasks' flow supply. Figure 5.20a show a construct that ensures at most one out of two tasks is placed. Figure 5.20b generalises the construct to n tasks and a maximum of p placed tasks.

"and" flow network construct Some tasks benefit if they are co-located with other tasks (e.g., tasks that exchange data regularly). Task affinity constraints (a type of complex co-location constraints) express requirements that must be satisfied for two or more tasks. It is impossible to represent such task requirements in min-cost flow networks, but it is possible to encode the them in generalised min-cost flow networks.

The generalised min-cost flow problem finds min-cost flow solutions in flow networks in which arcs have positive flow multipliers γ_{ij}, called gain factors. In these networks, each unit of flow sent across any arc (i, j) is transformed into γ_{ij} units of flow, which are receveived at node j. I use γ factors to build an "and" flow network construct to encode logical conjunction. In other words, the "and" network construct can be used to ensure that two or more tasks are only placed when they all respect a certain condition (e.g., all placed on the same machine).

In Figure 5.21, I show a network with two tasks to exemplify the network construct. In this example, task $T_{0,0}$ and $T_{0,1}$ must be either co-located or left unscheduled. I connect the task nodes to a new aggregator node G, which I then connect to node E and node A. I set a minimum flow requirement and a maximum capacity of one on arc (G,E), and I set a maximum capacity of one and a γ gain factor of two on arc (G,A). These arc requirements and capacities limit the possible task supply flow routes to two options:

1. The solver routes a task's flow supply via the unscheduled aggregator node U_0 and the

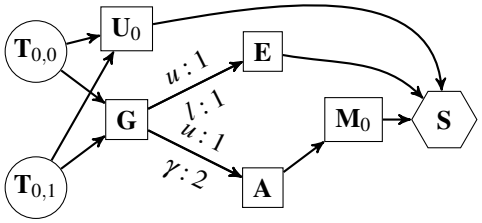


Figure 5.21: "and" flow network construct. It models tasks' requirement for being co-located on machine M_0.

other task's flow supply via node aggregator G. The flow routed via G must then be routed to node E because arc (G,E) has a minimum flow requirement of one. Thus, both tasks remain unscheduled.

2. The solver routes both tasks' flow supply to node aggregator G. Following, the solver routes exactly one unit of flow to node E because arc's (G,E) minimum flow requirement and maximum capacity are both one. The solver routes the remaining flow at node aggregator G to node A, but node A receives two units of flow because arc (G,A) has a γ gain factor equal to two. Thus, the solver routes two units of flow to machine M_0, and co-locates the tasks.

The "and" flow network construct in Figure 5.21 can be extended to n tasks. Like in the example, all tasks must be connected to node G, which must be connected to nodes E and A. The minimum flow requirement and maximum capacity on arc (G,E) must be set to $n-1$, and the gain factor on arc (G,A) must be set to n.

The min-cost flow algorithms I implemented in Flowlessly do not find optimal flow in generalised flow networks, but I plan to extend Flowlessly with generalised min-cost flow algorithms in the future.

5.4.3 Gang scheduling

Some cluster workflows expect all tasks to synchronise at certain computation points. For example, several graph processing back-end execution engines (e.g., Pregel, Giraph) synchronise all tasks at the end of each iteration (§ 2.1). Unless tasks are gang scheduled, workflow makespan increases and resources are wasted by tasks that wait at syncronisation points. Task gang scheduling is an instance of complex placement constraints; several tasks have mutual dependencies: they all must be placed or none. Despite the performance benefits, only two cluster schedulers (Tarcil [DSK15] and TetriSched [TZP+16]) can gang schedule tasks.

Schwarzkopf discusses gang scheduling in the context of min-cost flow schedulers [Sch16, §5.4.3]. He proposes leveraging minimum flow arc requirements to force tasks to schedule. In his approach, the scheduler adds per-job gang aggregator nodes to which it connects tasks it

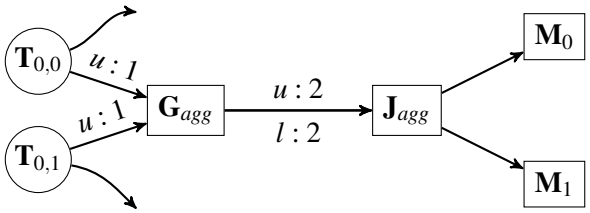


Figure 5.22: Example of how gang scheduling can be represented in a flow-network.

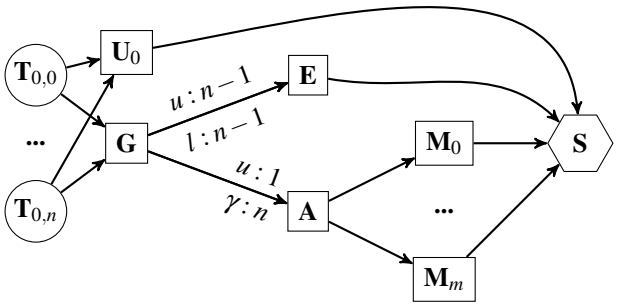


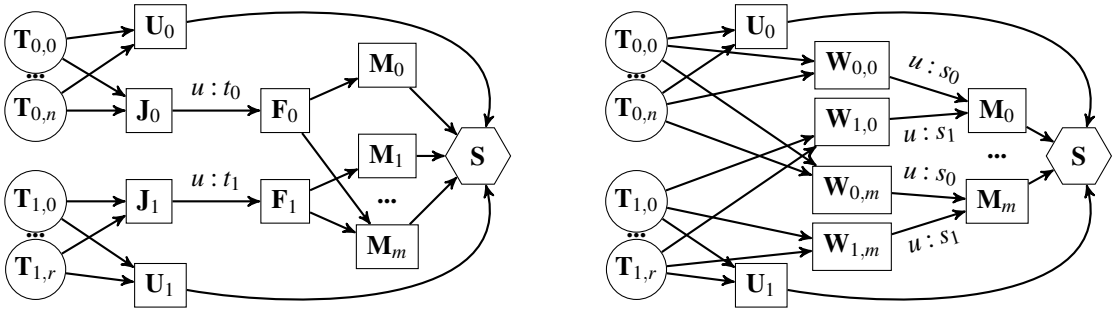
Figure 5.23: Representation of gang scheduling tasks in the flow network.

gang schedule. Following, the scheduler connects each gang aggregator node to a corresponding new per-job aggregator node. The scheduler sets a minimum flow requirement on the arc connecting each job's two aggregator nodes equal to the number of job tasks it must gang schedule. Finally, the scheduler adds arcs from job aggregator nodes to prefered resource nodes.

In Figure 5.22, I show an example of a flow network that encodes two tasks' gang scheduling constraints. The minimum flow requirement on arc ($\mathbf{G}_{agg}$, $\mathbf{J}_{agg}$) forces tasks' flow supply to traverse the arc. Thus, the scheduler is guaranteed to place both tasks on machines regardless of other available placement options.

However, Schwarzkopf's approach is limited: all tasks are placed no matter how costly it is, and how many tasks (potentially higher-priority tasks) must be preempted. This limitation exists because the minimum flow requirements constraints must be satisfied for the output flow to be optimal and feasible. One way to work around this limitation is to construct two flow networks: *(i)* a network that contains the gang scheduling construct, and *(ii)* a network without the construct. The scheduler could run two Flowlessly instances in parallel and pick the lowest cost solution. However, Firmament could simultaneously consider only few jobs for gang scheduling because it would have to execute 2^N solver instances in parallel (where N is the number of jobs).

I suggest an alternative approach for gang scheduling that does not have the above-mentioned limitations. My approach uses a generalised version of the "and" flow network construct (see Figure 5.23). I connect all n tasks that must be gang scheduled to aggregator node G. Like previously, I next connect node G to node E and node A, but now I set arc's (G,E) minimum flow requirement and maximum capacity to $n-1$, and arc's (G,A) gain factor to n. Thus, either all tasks remain unscheduled if the solver routes their flow supply to the sink node S along the



(a) Per-job token-based fairness.

(b) Worker-side weighted fair sharing.

Figure 5.24: Apollo-style per-job token-based fairness (5.24a) and Sparrow-style worker-side weighted fair sharing (5.24b) represented in flow networks.

unscheduled aggregator node U_0 and node E, or they are placed on machines if the solver routes one unit of their flow supply across arc (G,A).

5.4.4 Fairness

Data center operators strive to offer predictable performance and to fairly share resources among users (see § 2.3.1.7). I now introduce how different fairness types offered by state-of-the-art schedulers can be represented in flow networks, the abstraction Firmament uses.

5.4.4.1 Slot-based fairness

Many schedulers partition cluster machines into slots to which they assign tasks (e.g., Quincy, Sparrow). Each slot executes a single task at a time. These schedulers use admission control to enforce *slot-based* fair shares among users. For example, the Apollo scheduler uses a token-based mechanism that allocates capacity to jobs. Each job receives a fixed amount of tokens that it can spend to run tasks [BEL+14]. If a job exhausts its assigned tokens then it can run tasks opportunistically, but it must fairly share resources with other jobs that exhausted their tokens. The Sparrow distributed scheduler maintains for each job a worker-side queue on each cluster machine [OWZ+13]. The scheduler performs weighted fair queuing across machine queues. However, it ensures only weighted fair shares among jobs that run tasks on the same machines because Sparrow scheduler instances do not have information about all the tasks that run in a cluster.

The min-cost flow scheduling abstraction is general enough to represent the types of fairness Apollo and Sparrow support. In Figure 5.24a, I model Apollo's per-job token-based fairness in a flow network. I connect task nodes to per-job node aggregators (J_0 and J_1), which in turn I connect to a per-job fairness node aggregators (F_0 and F_1). For each job, I set a maximum capacity equal to the number of tokens a job is allocated on the arc connecting its corresponding aggregators (i.e., (J_0,F_0) and (J_1,F_1)). Neither job can execute more tasks than the number of

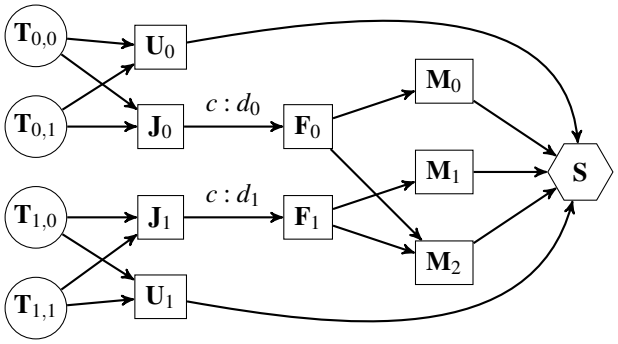


Figure 5.25: Dominant Resource Fairness represented in the flow network.

tokens it is allocated because its task nodes are connected only to the unscheduled and per-job aggregator nodes.

In Figure 5.24b, I model Sparrow's worker-side weighted fair sharing in the flow network. I introduce a node aggregator for each job and machine pair (e.g., $W_{1,m}$ for job 1 and machine m), and I connect each task to each node aggregator corresponding for its job. I also connect each node aggregator to its corresponding machine, and I set maximum capacities equal to weighted fair shares on these arcs. Thus, jobs cannot exceed their allocated weighted fair share on either machine.

5.4.4.2 Dominant resource fairness

Cluster workloads are diverse and have different resource requirements (§ 2.1). As a result, the cluster schedulers that achieve high resource utilisation bin-pack tasks on different resource dimensions (§ 2.3.1.3). However, the schedulers I discussed so far in this section consider treat tasks equally, and only provide slot-based fairness. In practice, these schedulers do not fairly share resources when tasks have diverse resource requirements.

Dominant Resource Fairness (DRF) [GZH+11; BCF+13] addresses this issue. It enforces max-min fairness across multiple resource dimensions. DRF considers a resource dominant for a user, if the resource has the highest allocated share across all the resources the user asks for. In each scheduling round, DRF places a task submitted by the user with the lowest dominant resource share, updates her resource shares and dominant resource if necessary. In Figure 5.25, I show dominant resource fairness is modelled in a flow network. I connect each job's task nodes to a per-job node aggregator (J_x), which in turn I connect to a fairness aggregator node (F_x). On each (J_x, F_x) arc, I set the cost (d_x) to how much job x's dominant resource share increases when the scheduler places one of its tasks. I also make sure to set other arc costs such that no task flow supply can be routed to the sink along a path with a cost smaller than double the minimum increase in dominant resource share across all jobs. This restriction ensures that the scheduler prioritises increasing the dominant resource shares over other objectives (e.g., data locality, avoiding task co-location interference).

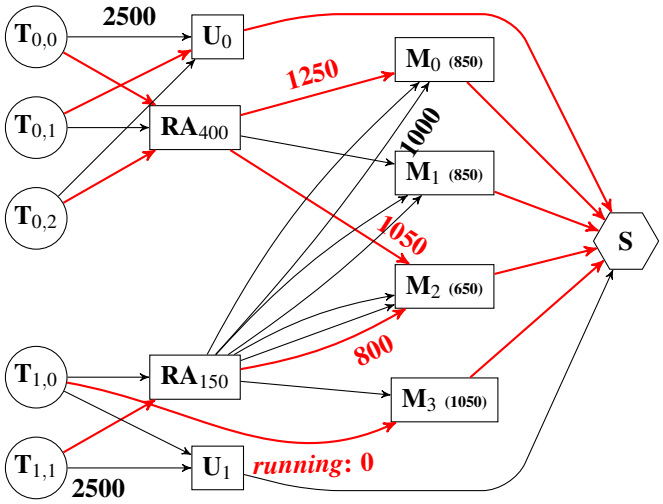


Figure 5.26: Example of a flow network for a network-aware policy with request aggregators (**RA**) and dynamic arcs to machines with spare network bandwidth.

5.5 Network-aware scheduling policy

Increasingly more data processing systems store data in memory and execute tasks that place high load on data center networks [ZCD+12; MMI+13]. Similarly, machine learning systems train models on large data using worker tasks that utilise a high fraction of machine network bandwidth. If these tasks are co-located, they interfere on the network, and thus take longer to complete. Moreover, they also can cause degradations in service task application-level performance (e.g., increase web serving latency) [GSG+15]. I introduce non-linear arc costs – one of the min-cost flow scheduling extensions I discussed – in a scheduling policy to showcase their practical application for network-intensive workloads.

In Figure 5.26, I illustrate a scheduling policy I developed, which avoids overcommitting machine network bandwidth (which degrades task makespan and application-level performance). Users specify task network bandwidth requests, which I use in this policy to avoid network interference. I create a request node aggregator (RA_x) if exists at least one task with a x network bandwidth request in the flow network. I connect each task node to the request aggregator node that corresponds to its network bandwidth request. Following, I add arcs from each RA_x node aggregator to machines on which there is at least as much network bandwidth available as RA's task request. I could use linear costs on each each arc (RA_x, M_m) – e.g, set arc cost to x, by how much network utilisation increases when a task with x network bandwidth request is placed on machine m. But the scheduling policy could then place many tasks on a machine during a scheduling round, instead of distributing tasks across the cluster. Instead, I use non-linear arc costs. I create as many arcs as tasks fit within the available machine network bandwidth between each RA aggregator and machine pair. I set the costs on these arcs to the sum of available machine network bandwidth, task bandwidth request and bandwidth requests of other tasks that are placed on this machine in the current scheduling round. For example, in Figure 5.26, machine M_2 has available 650 MB/s. I set the cost on the first arc between RA_{150}

and M_2 to $650 + 150$, the cost on the second arc to $650 + 150 * 2$, and the cost on the third arc to $650 + 150 * 3$. Thus, the scheduler would only place two tasks with a network request of 150 MB/s if there are no other machines with a network utilisation smaller than 750 MB/s. Firmament dynamically adapts these arcs as it observes bandwidth utilisation changes. The non-linear arc cost incentivise Firmament to balance machine network utilisation. In Section 6.3.2, I show that Firmament's network-aware scheduling policy outperforms other state-of-the-art policies on a network-intensive workload.

5.6 Limitations

Min-cost flow-based schedulers represent the scheduling problem as a min-cost flow optimisation. These flow network representation can model many desirable scheduler features (e.g., soft and hard constraints, data locality), but it also has several limitations that I discuss now.

Multi-dimensional capacities Scheduling policies restrict flow paths by setting minimum flow requirements and maximum capacity constraints on arcs. However, despite tasks having widely different requirements on multiple resource dimensions (see § 2.1), min-cost flow solvers do not distinguish between different task flow supplies. Solvers ensure that each unit of flow satisfies the same minimum flow requirements and capacity constraints as other flow, regardless of the resource requirements the task that generates it has. As a result, min-cost flow schedulers admission control tasks to the flow network (i.e., they add tasks only when enough resources are available). Multi-dimensional task resource requests could be expressed in more general flow networks which route flow for multiple commodities (i.e., tasks create flow supply in different dimensions and arcs have capacity constraints for each dimension). However, finding the minimum cost flow for a multi-commodity network is a NP-complete problem.

Multi-dimensional arc costs Flow network arc costs represent how expensive is to route task flow along arcs. For example, the cost of an arc that connects a task node to a machine node represents how expensive it is place the task on the machine. Regardless of resource requirements tasks have in many resource dimensions, a task placement cost must be represented as an arc cost or a sum of arc costs. Min-cost flow schedulers use weighted linear functions to flatten different task resource requirements to integer arc costs. However, it is challenging for the developer of a scheduling policy to define these functions such that they accurately predict how well-suited machines are for given tasks. This weighted linear functions would not be required if min-cost flow schedulers were to generate multi-commodity flow networks, but as I previously noted, it is not possible to quickly find solutions for such flow networks.

5.7 Summary

In this chapter, I argue that min-cost flow-based centralised schedulers can be optimised to achieve low task placement latency without decreasing placement quality.

I first describe how min-cost flow-based schedulers work, compare them to traditional task-by-task schedulers, and show that current min-cost flow schedulers do not scale to large clusters (§5.1). Following, I introduce Firmament, a min-cost flow scheduler I extend. I describe how one implements scheduling policies in Firmament, and how the scheduler interacts with min-cost flow solvers (§5.2). Next, I describe and compare different min-cost flow algorithms, discuss edge cases, and techniques I developed for my min-cost flow solver to provide low algorithm runtime in all cases (§5.3). In section 5.4, I describe several scheduling features that previously were thought to be expensive or impossible to express in min-cost flow-based schedulers. Subsequently, I use one of these features to build a new scheduling policy that offers high-quality placements (§5.5). Finally, I highlight min-cost flow schedulers limitations and discuss several ways to address them (§5.6).

In Chapter 6, I show that with my extensions Firmament places tasks in hundreds of milliseconds on 12,500-machine clusters. Moreover, I show using a mixed-task workload that Firmament chooses better placements than state-of-the-art distributed and centralised schedulers.

Chapter 6

Evaluation Firmament

Firmament's goals are to choose high-quality placements with low task scheduling latency even on large clusters. In this chapter, I show that Firmament meets these goals by running a range of large cluster simulations and experiments on a 40-machine cluster using real-world workloads.

I first focus on evaluating Firmament's scalability (§6.2). Following, I compare Firmament's placements with those made by several state-of-the-art centralised and distributed schedulers (§6.3). With my experiments, I answer the following questions:

1. Does Firmament scale better than Quincy on large clusters when applying the same scheduling policy? (§6.2.1)
2. How well does Firmament handle challenging scheduling situations (e.g., oversubscribed clusters)? (§6.2.2)
3. How does Firmament's placement latency scale for short tasks, and where is Firmament's breaking point? (§6.2.3)
4. Is Firmament able to offer low task placement latency to future workloads in which task runtimes are going to decrease? (§6.2.4)
5. How does Firmament's scalability affect placement quality? (§6.3.1)
6. How well does Firmament work compared to other schedulers on a homogeneous cluster running a real-world mix of short analytics tasks and long-running batch and service jobs? (§6.3.2)

6.1 Experimental setup

My experiments combine scale-up simulations with experiments on a local homogeneous testbed cluster. In **simulations**, I replay a public production workload trace from a 12,500-machine

	Machine	Architecture	Cores	Threads	Clock	RAM
40×	Dell R320	Intel Xeon E5-2430Lv2	6	12	2.4 GHz	64 GB PC3-12800

Table 6.1: Specifications of the machines in the local homogeneous cluster.

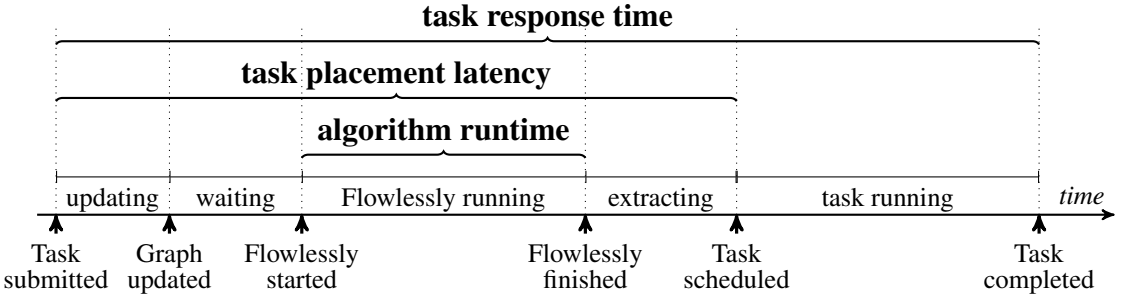


Figure 6.1: The metrics I measure shown with respect to a task's scheduling stages.

Google cluster [RTG+12] against Firmament's implementation. My simulator uses a similar design to Borg's "Fauxmaster" [VPK+15, §3.1] and to Kubernetes's "Kubemark" [KBM17]: it merely stubs out RPCs and task execution, but otherwise runs the real code and scheduling logic against simulated machines. However, there are three important methodological limitations to note. First, the Google trace contains multi-dimensional resource requests for each task. Firmament supports multi-dimensional feasibility checking (as in Borg [VPK+15, §3.2]), but in order to fairly compare to Quincy, I use slot-based placement. Second, I do not enforce task constraints for the same reason, despite them helping Firmament's min-cost flow solver to choose placements. Third, the Google trace does not contain information about job types or input sizes. I use the same priority-based job type classification as in Omega [SKA+13], and estimate batch task input sizes as a function of the known runtime according to typical industry distributions [CAK12].

In all the experiments I compare with **Quincy**, I use my implementation of Quincy's scheduling policy on Firmament, and run the cost scaling min-cost flow algorithm only, as in Quincy.

In **local cluster experiments**, I use the homogeneous 40-machine cluster I describe in Table 6.1. The machines are distributed across four racks and are connected by a 10G network in a leaf-spine topology. The core interconnect offers a 320 Gbit/s bandwidth. All machines run Ubuntu 14.04 (Trusty Tahr) with Linux kernel v3.14 and are included in a Hadoop File System (HDFS) deployment. The local cluster models a small to medium cluster, albeit a fully controlled environment without any external network traffic or machine utilisation. I use the local cluster to accurately measure Firmament's scheduling quality. I do not use a larger EC2 cluster because my tasks can interfere with the workloads other tenants run. As a result, I would not be able to isolate task interference caused by Firmament's placements from interference generated by other tenants.

6.1.1 Metrics

In Figure 6.1, I show the task metrics I focus on in my evaluation and highlight how they relate to Firmament's scheduling stages. **Algorithm runtime** is the time it takes Flowlessly to run the best min-cost flow algorithm and represents how much time a task spends being actively scheduled.

Task placement latency represents the time between task submission and task placement. This metric includes task wait time, the time Firmament spends updating the flow network, Flowlessly's runtime and the time it takes to extract task placements from the optimal flow. It is important to include task wait time in the latency metric because min-cost flow-based schedulers reconsider the entire existing workload each time they run a min-cost flow algorithm (see Section 5.1). These schedulers cannot place any tasks while the min-cost flow solver runs. Thus, tasks may have to wait for up to a min-cost flow solver run until they are considered by a solver.

Finally, I measure **task response time**[1] to quantify Firmament's placements quality. Task response time is the total time between task submission and task completion. It is an end-to-end metric that captures how quickly the scheduler places tasks and how good these placements are. High placement quality increases cluster utilization and avoids performance degradation due to overcommit. However, high-quality placements may not decrease task response time if the scheduler cannot choose them with low placement latency. Poor placement quality, by contrast, decreases application level performance (for long-running services), or increases task response time (for batch and interactive tasks).

6.2 Scalability

In my first set of experiments, I evaluate Firmament's scalability. I compare Firmament's task placement latency to Quincy's (§6.2.1), I study how fast Flowlessly finds the optimal min-cost flow in challenging cluster situations (§6.2.2), I measure Firmament's scalability to very short tasks (§6.2.3), and I evaluate how well Firmament can handle future workloads with shorter tasks (§6.2.4).

6.2.1 Scalability *vs.* Quincy

In Figure 5.5, I illustrate that Quincy fails to scale to clusters of thousands of machines at acceptable task placement latencies. I now repeat the same experiment using Firmament on the full Google-scale simulated cluster. I measure *task placement latency*, i.e., the time between

[1] Task response time is primaly meaningful for batch and interactive data processing tasks; long-running service tasks' response time are conceptually infinite, and in practice are determined by failures and operational decisions.

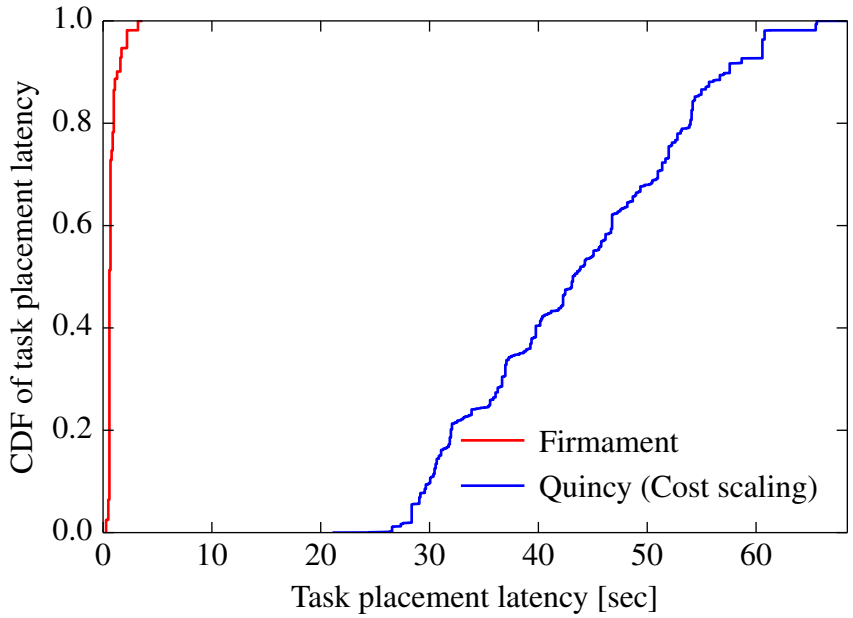


Figure 6.2: Firmament achieves 20× lower task placement latency than Quincy on a simulated 12,500-machine cluster at 90% slot utilisation, replaying the Google trace. The scheduling quality is unaffected.

task submission and task placement. In comparison to Figure 5.5, I increase cluster slot utilisation to 90% (i.e., I decrease the number of slots per-machine, but I do not change the cluster workload trace) to make the setup more challenging for Firmament. I also tune Quincy's cost scaling-based min-cost flow solver for best performance.[2]

In Figure 6.2, I show the results as a CDF of task placement latency. Quincy takes between 25 and 60 seconds to place tasks, whereas, Firmament typically places tasks in hundreds of milliseconds and only exceeds a 1-second placement latency in the 97th percentile. This is more than a 20× improvement over Quincy's placement latency without any reduction in placement quality. Firmament finds the same optimal placements as Quincy does and meets my goal of low task placement latency on large clusters.

6.2.2 Scalability in extreme situations

In the previous experiment, Firmament chooses placements fast because it uses the relaxation algorithm that handles the Google cluster workload well. However, in Subsection 5.3.2 I describe a cluster setup in which relaxation does not work well. In this cluster setup, Firmament's solver, Flowlessly, automatically falls back to the incremental cost scaling algorithm if it is faster (§5.3.6). I now show the benefits of running two algorithms rather than just one.

I run an experiment in which I shrink the number of slots for each cluster machine. The average

[2]Specifically, I found that an α-factor parameter value of 9, rather than the default of 2 used in Quincy, improves runtime by ≈30%.

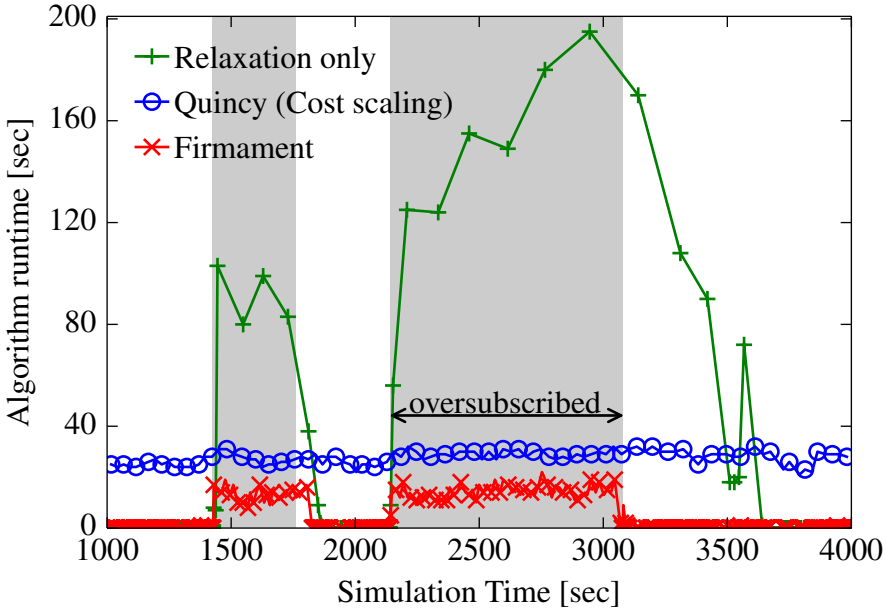


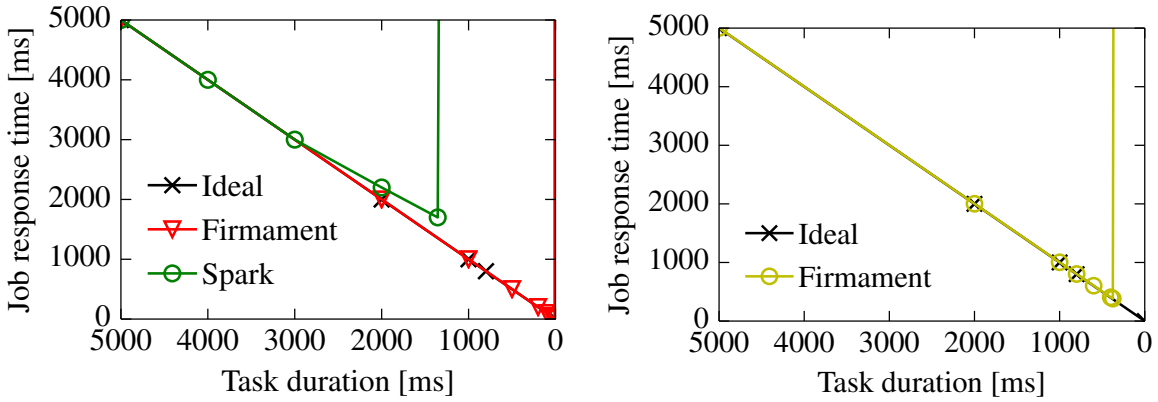
Figure 6.3: Relaxation's runtime increases when the cluster is oversubscribed (grey areas). Firmament handles well oversubscription. It runs 2×faster than Quincy's cost scaling. Moreover, it recovers 500s earlier from task backlog.

cluster slot utilisation is 97%, but the cluster also experiences transient periods of oversubscription. In Figure 6.3, I compare Flowlessly's automatic use of the fastest algorithm against using only one algorithm, either relaxation or cost scaling. During oversubscription (highlighted in grey), relaxation's runtime spikes to hundreds of seconds per run, while cost scaling alone completes in ≈30 seconds independent of cluster load. Flowlessly correctly falls back to incremental cost scaling in this situation, the algorithm that finishes first. It takes 10–15 seconds to complete, which is about 2× faster than using cost scaling only (as Quincy does).

Moreover, Flowlessly recovers earlier from the cluster oversubscription starting at 2,200s: relaxation runtime returns to sub-second levels at around 3,700s, whereas Flowlessly returns at around 3,100s.

Relaxation on its own takes longer to recover because no placement decisions are made while the algorithm runs. The slots freed by tasks that complete are not re-used only once the following solver runs complete, even though new, waiting tasks accumulate. Thus, the scheduler underutilises the cluster when it uses relaxation only, despite that some tasks wait to be scheduled.

To sum up, my experiment shows that Flowlessly's combination of algorithms outperforms either algorithm running alone. As a result, Firmament obtains higher utilisation and offers lower task placement latency because it uses incremental cost scaling when the cluster is oversubscribed, and quickly returns to relaxation when utilisation decreases.



(a) 100 machines

(b) 1000 machines

Figure 6.4: Firmament fails to keep up when tasks are shorter than ≈5ms at 100-machine scale, and ≈375ms at 1,000-machine scale, at 80% cluster slot utilisation.

6.2.3 Scalability to very short tasks

Task throughput and task durations from the Google trace workload are challenging for Quincy, but they are insufficient to stress Firmament. I now investigate Firmament's min-cost flow-based approach scalability limits in the absence of oversubscription. I subject Firmament to a worst-case workload consisting entirely of a huge number of short tasks in order to find its breaking point. This experiment is similar to Sparrow's breaking-point experiment for the centralised Spark scheduler [OWZ+13, Fig. 12].

In the experiment, I simulate in turn three clusters of increasing size: 100, 1,000 and 10,000 machines. I submit jobs composed of 10 tasks at an interarrival time that keeps the cluster at a constant load of 80% if there is no scheduler overhead. I measure *job response time*, which is the maximum of the ten task response times for a job. If tasks schedule immediately and as a wave, the job response time is equal to the tasks' runtime. I designed this experiment to only measure scheduler scalability, thus I do not increase response time for tasks that are placed on highly-utilised machines (i.e., tasks do not interfere).

In Figure 6.4, I plot job response time as a function of decreasing task duration. As I reduce task duration, I also reduce job interarrival time to keep the load constant, hence increasing the task throughput faced by the scheduler. Task-by-task schedulers (e.g., Spark's scheduler, Sparrow) are at advantage in this experiment because they can quickly make a decision for each incoming task. By contrast, Firmament runs a min-cost flow optimisation over the entire workload (including running tasks) each time.

The experiment stresses both scheduler task placement throughput and task placement latency. A scheduler that provides high task placement throughput by batching task and amortising work may not keep up with the workload because of low job interarrival times. Many tasks complete while the scheduler runs, but a batching scheduler would not reuse the freed resources until the next run. In the experiment, with an ideal scheduler, job response time would be equal to task

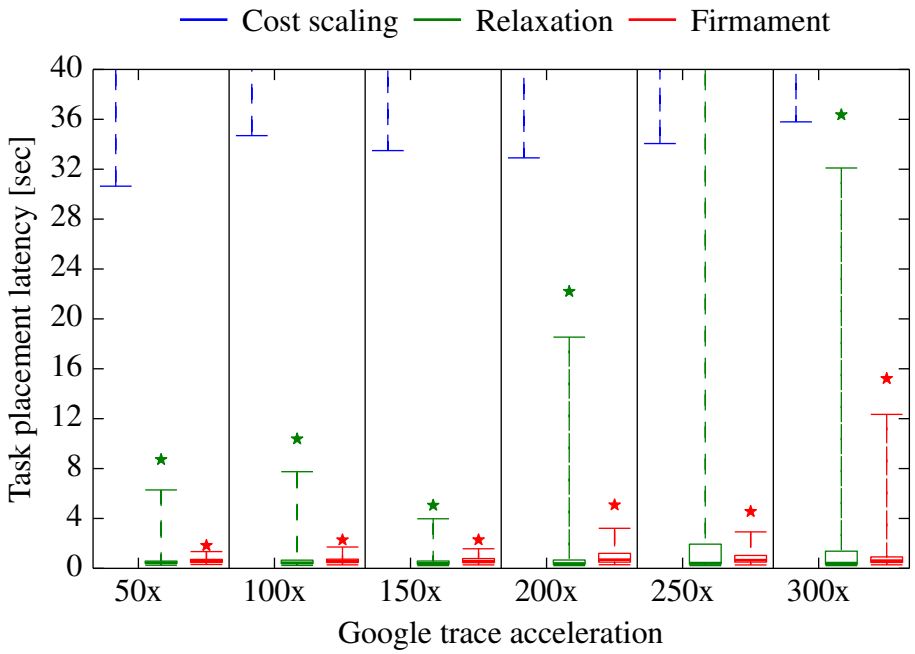


Figure 6.5: Firmament can schedule a 300× accelerated Google workload, while using relaxation only achieves far poorer placement latencies in the tail.

duration because the scheduler would take no time to choose placements. But in practice, the higher scheduler's task placement latency is, the sooner job response time deviates from the diagonal. The breaking point occurs when the scheduler's placement latency is at least as high as it takes to receive enough tasks to utilise the remainder available resources (i.e., 20% of the cluster). At that point, the scheduler accumulates an ever growing backlog of unscheduled tasks. This is not the case with the Sparrow distributed scheduler. It achives job response times that are close to ideal on the 100 machine cluster [OWZ+13, Fig. 12]. However, centralised schedulers struggle with this workload. For example, Spark's centralised task scheduler in 2013 had its breaking point on 100 machines at a 1.35 second task duration [OWZ+13, §7.6].

By contrast, even though the centralised Firmament scheduler runs a min-cost flow optimisation over the entire workload every time, Figure 6.4 shows that it keeps up with the workload and achieves near-ideal job response time down to task durations as low as 5ms (100 machines) or 375ms (1,000 machines). This makes Firmament's response time competitive with distributed schedulers on medium-sized clusters that only run short interactive analytics tasks. At 10,000 machines, Firmament keeps up with task durations ≥5s. However, such large clusters do not typically run short tasks only, but a mix of long-running and short tasks [BEL+14; DDK+15; KRC+15; VPK+15]. I therefore next investigate Firmament's performance on a realistic mixed cluster workload.

6.2.4 Scalability to future workloads

In this experiment, I simulate the workload of a 12,500-machine cluster using the Google trace. However, I accelerate the trace by dividing all task runtimes and interarrival times by a acceleration factor. This simulates a future workload that consists of shorter batch tasks [OPR+13], while service jobs continue to be long-running. For example, with a 300× acceleration, the median batch task takes 1.4 seconds, and the 90th and 99th percentile batch tasks take 12 and 61 seconds.

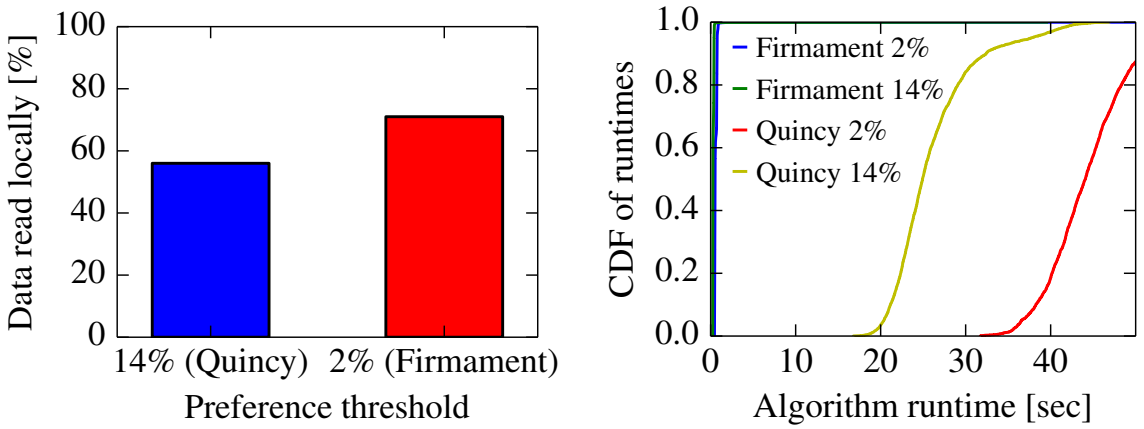
In Figure 6.5, I compare Firmament's placement latency to the ones offered by individual min-cost flow algorithms. I measure placement latency across all tasks, and plot distributions (1st, 25th, 50th, 75th, 99th percentile, and maximum). As before, a single min-cost flow algorithm does not scale: cost scaling's placement latency already exceeds 30 seconds even with a 50× acceleration, and relaxation sees tail placement latencies well above 10 seconds beyond a 150× acceleration. Whereas, even at a acceleration of 300×, Firmament keeps up and places 75% of the tasks at with sub-second latency. Hybrid schedulers [DDK+15; KRC+15; DDD+16] could probably support these future workloads, but in contrast to Firmament they would sacrifice placement quality for short tasks and cause interference for long tasks.

6.3 Scheduling quality

I now evaluate Firmament's placement quality. I first compare Firmament to Quincy and show that my scheduler not only offers lower task placement latency, but it also increases the fraction of input data that is locally read (§6.3.1). Finally, I evaluate Firmament on a mixed cluster workload and show that it outperforms three state-of-the-art centralised schedulers and one distributed scheduler (§6.3.2).

6.3.1 Improving data-locality

Due to the scalability improvements I made, Firmament can use more complex scheduling policies that generate large flow networks. In this experiment, I evaluate what effect has increasing the number of arcs in the flow network on the placement quality. I use the Google cluster trace to simulate a 12,500 machine cluster, and as an illustrative example, I vary the data locality threshold in the Quincy scheduling policy. This threshold decides what fraction of a task's input data must reside on a machine or within a rack in order for the former to receive a preference arc to the latter. Quincy originally picked a threshold of a maximum of ten arcs per task. However, in Figure 6.6b I show that even a higher threshold of 14% local data, which corresponds to at most seven preference arcs, yields algorithm runtimes of 20–40 seconds for Quincy's cost scaling.



(a) % of data read locally

(b) Algorithm runtime

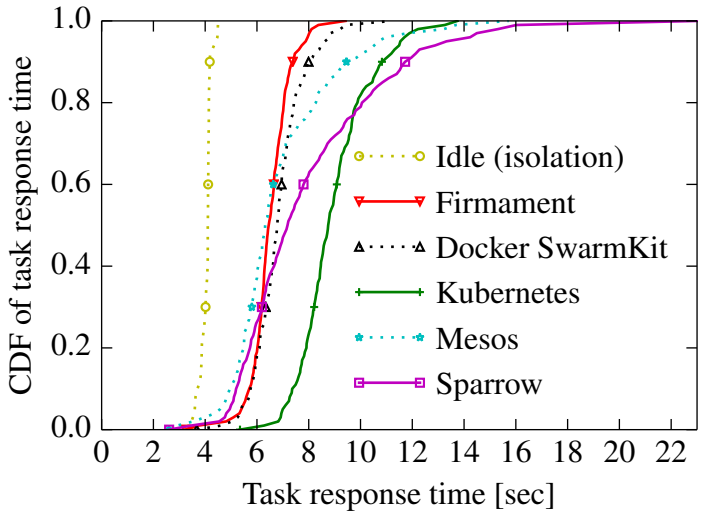
Figure 6.6: Firmament achieves 27% higher task data locality than Quincy (6.6a). This comes without any placement latency increase because Firmament's min-cost flow solver has smaller runtime even when I lower the task data locality preference threshold to 2% (6.6b).

A low threshold allows the scheduler exploit more fine-grained locality, but increases the number of arcs in the graph. Consequently, in Figure 6.6a, I show that if I lower the threshold to 2% local data, the percentage of locally read input data increases from 56% to 71%, which saves 4 TB of network traffic per simulated hour. Firmament is required to achieve this benefit; Figure 6.6a illustrates that when I use a 2% locality threshold in Quincy, algorithm runtime doubles, while Firmament still achieves almost the same algorithm runtime as with a 14% threshold.

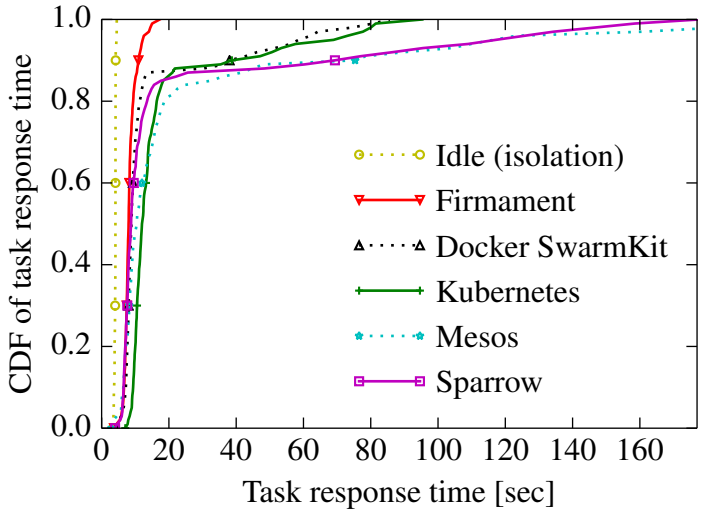
6.3.2 Network-aware scheduling

I now evaluate how good Firmament's placements are compared to other centralised and distributed schedulers. I deploy Firmament on the local 40-machine homogeneous cluster (Table 6.1) to evaluate its performance with real cluster workloads. I run a workload of short interactive data processing tasks that take 3.5–5 seconds to complete on an otherwise idle cluster. Each task reads inputs of 4–8 GB from a cluster-wide HDFS installation, and in this experiment I use Firmament's network-aware scheduling policy. This policy reflects current network bandwidth reservations, considers actual bandwidth usage in the flow network's costs, and strives to place tasks on machines with lightly-loaded network connections.

In Figure 6.7a, I show CDFs of task response times I obtained using different cluster managers' schedulers. I compare to a baseline that runs one by one each task in isolation on an otherwise idle cluster and network. Firmament achieves the closest task respone time to the idle (isolation) baseline in the tail (80th percentile upwards) as it successfully avoids overcommitting machines' network bandwidth. Other schedulers make random placements (Sparrow), are effectively random as they do not take network bandwidth into account (Mesos, Kubernetes), or perform simple load-spreading based on the number of running tasks (Docker SwarmKit).



(a) Short interactive data processing tasks running on a cluster with an otherwise idle cluster and network. Overhead over "idle" due to network contention.



(b) Short interactive data processing tasks running on a cluster with background traffic from long-running batch and service tasks.

Figure 6.7: On a local 40-node cluster, Firmament reduces task response time of short batch tasks in the tail using a network-aware scheduling policy, both (a) without and (b) with background traffic. Note the different *x*-axis scale.

Real-world clusters, however, run a mix of short interactive data processing tasks, long-running services and batch processing tasks (§2.1). I therefore extend the workload with long-running batch and service jobs to represent a similar mix. The long-running batch tasks are generated by fourteen `iperf` clients who communicate using UDP with seven `iperf` servers. Each `iperf` client generates 4 Gbps of sustained network traffic and simulates the network pattern of a machine learning (e.g., TensorFlow [ABC+16]) worker that communicates with a parameter server (i.e., one of the `iperf` servers) in a higher-priority network service class than the short

batch tasks. Finally, I deploy three `nginx` web servers and seven HTTP clients as long-running service jobs. I run the cluster at about ≈80% network utilisation, and again measure the task response time for the interactive data processing tasks.

In Figure 6.7b, I show that Firmament's network-aware scheduling policy substantially improves the tail of the task response time distribution of interactive data processing tasks. For example, Firmament's 99th percentile response time is 3.4× better than the Docker SwarmKit and Kubernetes ones, and 6.2× better than Sparrow's. The tail matters, since the last task's response time often determines a batch job's overall response time (the "straggler" problem). Since this cluster is small, Firmament's task placement latency is inconsequential at around 5ms.

6.4 Summary

In this chapter, I investigated Firmament's scalability and placements quality. My experiments show that Firmament:

1. **Scales to large clusters:** Firmament maintains the same placement quality as Quincy and achieves sub-second task placement latency in the common-case on a 12,500-machine cluster (§6.2.1).

2. **Copes well with extreme cluster situations:** Firmament's min-cost flow solver completes in at most 17 seconds, by contrast individual min-cost flow algorithms complete at best in 25 seconds in any situation (cost scaling), or fail to keep up in challeging situations (e.g., relaxation in oversubscribed clusters) (§6.2.2).

3. **Scales to short tasks-only workloads:** Firmament has task placement latency comparable to the Sparrow distributed scheduler on a 1,000-machine cluster when scheduling workloads comprising of 375ms long tasks. However, Firmament fails to keep up with task lengths below 5s on 10,000-machine clusters (§6.2.3).

4. **Efficiently handles future workloads:** Firmament places tasks with low latency even on a 300× accelerated Google workload from a production cluster with a 1.4 seconds median task duration (§6.2.4).

5. **Improves Quincy's placements:** Firmament can use more complex scheduling policies because of its low task placement latency (e.g., add more preference arcs to the flow network). Thus, it increases the percentage of locally read input data from 56% (with Quincy) to 71% (§6.3.1).

6. **Outperforms centralised and distributed schedulers:** Firmament reduces the response time of short network-intensive interactive tasks by up to 6× compared to other schedulers on a cluster running a real-world mixed task workload (§6.3.2).

In my experiments, Firmament chooses high-quality placements as good as advanced centralised schedulers, but at the speed and scale typically associated with distributed schedulers. However, there are workloads and situations in which Firmament does not choose quality placements. In Chapter 7, I discuss these situations, Firmament's limitations and how these can be addressed.

Chapter 7

Conclusions and future work

I described a novel approach to building data processing systems and presented a new cluster scheduler that makes workflows more predictable and reduces their makespan.

- In **Chapter 3**, I presented Musketeer, a workflow manager that decouples front-end frameworks from back-end execution engines. Musketeer translates workflows expressed in front-end frameworks into an intermediate representation based on relational algebra operators. Following, it applies optimisations on the intermediate representation and uses several techniques such as operator merging and type inference to generate efficient code. Musketeer also can automatically map workflows to back-ends using an optimal, but slow algorithm, or a fast heuristic.
- In **Chapter 4**, I shown that Musketeer automatically generates efficient code that is at most 30% slower than hand-written optimised workflow implementations. I also demonstrated that Musketeer can reduce workflows' makespan by mapping them to a suitable back-end or combinations of back-ends. Finally, I evaluated Musketeer's automated mapping solution and shown that when it has access to entire workflow history, it is always choosing mappings that are within 10% of the best mapping for the tested workflows.
- In **Chapter 5**, I presented a centralised scheduler that makes high-quality placements, scales to large clusters and, chooses placements at the speed typically associated with distributed schedulers. To achieve this, Firmament uses the Flowlessly min-cost max-flow solver that combines multiple algorithms and relies on a range of techniques to speed up incremental optimisation of cluster scheduling placements.
- In **Chapter 6**, I demonstrated using a trace of a 12,500 machine cluster, that Firmament can place tasks within hundreds of milliseconds at scale. I also shown using a 250× accelerated cluster trace, that Firmament can place tasks with low scheduling latency even for future workloads in which tasks are shorter. Finally, I shown that Firmament makes better decisions than state-of-the-art centralised and distributed schedulers on a local 40-machine cluster.

In the remainder of this chapter, I discuss possible extensions to Musketeer

7.1 Improving Musketeer

7.1.1 More expressive intermediate representation

7.1.2 Beefy UDFs

7.2 Improving Firmament

7.2.1 Combining network and task scheduling

7.2.2 Apply machine learning to improve scoring

7.3 Summary

In this dissertation, I have presented

Bibliography

[AAK+11] Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert Greenberg, Ion Stoica, Duke Harlan, and Ed Harris. "Scarlett: Coping with Skewed Content Popularity in MapReduce Clusters". In: *Proceedings of the 6th European Conference on Computer Systems (EuroSys)*. Salzburg, Austria, Apr. 2011, pp. 287–300 (cited on pages 49, 148).

[ABB+13] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, et al. "MillWheel: Fault-tolerant Stream Processing at Internet Scale". In: *Proceedings of the VLDB Endowment* 6.11 (Aug. 2013), pp. 1033–1044 (cited on page 30).

[ABC+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. "TensorFlow: A system for large-scale machine learning". In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Savannah, Georgia, USA, Nov. 2016 (cited on pages 50, 169).

[ABE+14] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, et al. "The Stratosphere platform for big data analytics". In: *Proceedings of the VLDB Endowment* 23.6 (2014), pp. 939–964 (cited on page 34).

[AGO+92] Ravindra K. Ahuja, Andrew V. Goldberg, James B. Orlin, and Robert E. Tarjan. "Finding minimum-cost flows by double scaling". In: *Mathematical programming* 53.1-3 (1992), pp. 243–266 (cited on page 129).

[AGS+11] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. "Disk-locality in Datacenter Computing Considered Irrelevant". In: *Proceedings of the 13th USENIX Workshop on Hot Topics in Operating Systems (HotOS)*. Napa, California, USA, May 2011, pp. 12–17 (cited on page 49).

[AGS+13] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. "Effective Straggler Mitigation: Attack of the Clones". In: *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Lombard, IL, USA, 2013, pp. 185–198 (cited on page 49).

[AH00] Ron Avnur and Joseph M. Hellerstein. "Eddies: Continuously Adaptive Query Processing". In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Dallas, Texas, USA, 2000, pp. 261–272 (cited on page 67).

[AKB+12a] Sameer Agarwal, Srikanth Kandula, Nico Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. "Reoptimizing Data Parallel Computing". In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. San Jose, California, USA, 2012, pp. 281–294 (cited on page 77).

[AKB+12b] Sameer Agarwal, Srikanth Kandula, Nico Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. "Re-Optimizing data-parallel computing". In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. San Jose, California, USA, Apr. 2012, pp. 281–294 (cited on pages 86, 121).

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice Hall, 1993 (cited on pages 125, 127, 130).

[AU79] Alfred V. Aho and Jeffrey D. Ullman. "Universality of Data Retrieval Languages". In: *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*. San Antonio, Texas, 1979, pp. 110–119 (cited on page 76).

[AXL+15] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, et al. "Spark SQL: Relational Data Processing in Spark". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Melbourne, Victoria, Australia, 2015, pp. 1383–1394 (cited on pages 16, 29, 38, 67).

[Bac15] Emilian D. Bacila. "Planchet: Decoupling graph computations with user-defined functions". Computer Science Tripos Part II Dissertation. University of Cambridge Computer Laboratory, May 2015 (cited on page 79).

[BBD05] Shivnath Babu, Pedro Bizarro, and David DeWitt. "Proactive Re-optimization". In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Baltimore, Maryland, 2005, pp. 107–118 (cited on page 67).

[BBJ+14] Yingyi Bu, Vinayak Borkar, Jianfeng Jia, Michael J. Carey, and Tyson Condie. "Pregelix: Big(Ger) Graph Analytics on a Dataflow Engine". In: *Proceedings of the VLDB Endowment* 8.2 (Oct. 2014), pp. 161–172 (cited on pages 39, 74, 77, 82).

[BCF+13] Arka A. Bhattacharya, David Culler, Eric Friedman, Ali Ghodsi, Scott Shenker, and Ion Stoica. "Hierarchical Scheduling for Diverse Datacenter Workloads". In: *Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC)*. Santa Clara, California, Oct. 2013, 4:1–4:15 (cited on pages 55, 155).

[BCG+11] Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. "Hyracks: A flexible and extensible foundation for data-intensive computing". In: *Proceedings of the 27th IEEE International Conference on Data Engineering (ICDE)*. Apr. 2011, pp. 1151–1162 (cited on pages 29, 32, 34, 39).

[BCH13] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. "The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second edition". In: *Synthesis Lectures on Computer Architecture* 8.3 (July 2013), pp. 1–154 (cited on page 46).

[BCK+11] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. "Towards Predictable Datacenter Networks". In: *Proceedings of the 2011 ACM SIGCOMM Conference (SIGCOMM)*. Toronto, Ontario, Canada, 2011, pp. 242–253 (cited on page 53).

[BEL+14] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, et al. "Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing". In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, Colorado, USA, Oct. 2014, pp. 285–300 (cited on pages 26, 53–55, 57–58, 69, 116, 132, 154, 166).

[BHB+10] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. "HaLoop: Efficient Iterative Data Processing on Large Clusters". In: *Proceedings of the VLDB Endowment* 3.1-2 (Sept. 2010), pp. 285–296 (cited on pages 31–32, 34).

[BKV08] Robert M. Bell, Yehuda Koren, and Chris Volinsky. *The BellKor solution to the Netflix prize*. Technical report. AT&T Bell Labs, 2008 (cited on page 99).

[BT88a] Dimitri P. Bertsekas and Paul Tseng. "Relaxation Methods for Minimum Cost Ordinary and Generalized Network Flow Problems". In: *Operations Research* 36.1 (Feb. 1988), pp. 93–114 (cited on page 128).

[BT88b] Dimitri P. Bertsekas and Paul Tseng. "The Relax codes for linear minimum cost network flow problems". In: *Annals of Operations Research* 13.1 (Dec. 1988), pp. 125–190 (cited on page 128).

[CAB+12] Yanpei Chen, Sara Alspaugh, Dhruba Borthakur, and Randy Katz. "Energy Efficiency for Large-scale MapReduce Workloads with Significant Interactive Analysis". In: *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*. Bern, Switzerland, Apr. 2012, pp. 43–56 (cited on pages 47, 52).

[CAK12] Yanpei Chen, Sara Alspaugh, and Randy Katz. "Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads". In: *Proceedings of the VLDB Endowment* 5.12 (Aug. 2012), pp. 1802–1813 (cited on pages 40, 68, 76, 161).

[CBB+13] Michael Curtiss, Iain Becker, Tudor Bosman, Sergey Doroshenko, Lucian Grijincu, Tom Jackson, Sandhya Kunnatur, et al. "Unicorn: A System for Searching the Social Graph". In: *Proceedings of the VLDB Endowment* 6.11 (Aug. 2013), pp. 1150–1161 (cited on page 36).

[CJL+08] Ronnie Chaiken, Bob Jenkins, Per-Ake Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. "SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets". In: *Proceedings of the VLDB Endowment* 1.2 (Aug. 2008), pp. 1265–1276 (cited on pages 29, 38, 69).

[CLL+11] Biswapesh Chattopadhyay, Liang Lin, Weiran Liu, Sagar Mittal, Prathyusha Aragonda, Vera Lychagina, Younghee Kwon, et al. "Tenzing: A SQL Implementation On The MapReduce Framework". In: *Proceedings of the 37th International Conference on Very Large Data Bases (VLDB)*. Seattle, Washington, USA, Aug. 2011, pp. 1318–1327 (cited on pages 29, 38, 69).

[Cod70] Edgar F. Codd. "A Relational Model of Data for Large Shared Data Banks". In: *Communications of the ACM* 13.6 (June 1970), pp. 377–387 (cited on pages 38, 67, 76).

[CRP+10] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. "FlumeJava: Easy, Efficient Data-parallel Pipelines". In: *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Toronto, Ontario, Canada, June 2010, pp. 363–375 (cited on pages 29, 39, 44, 67, 77, 89).

[CSC+15] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. "PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs". In: *Proceedings of the 10th ACM European Conference on Computer Systems (EuroSys)*. Bordeaux, France, Apr. 2015 (cited on page 37).

[CSM13] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. "Optimizing Database-backed Applications with Query Synthesis". In: *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Seattle, Washington, USA, 2013, pp. 3–14 (cited on pages 69, 91).

[CWI+16] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, et al. "Realtime Data Processing at Facebook". In: *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. San Francisco, California, USA, 2016, pp. 1087–1098 (cited on page 30).

[DDD+16] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. "Job-Aware Scheduling in Eagle: Divide and Stick to Your Probes". In: *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)*. Santa Clara, California, USA, Oct. 2016 (cited on pages 55, 59, 116, 167).

[DDK+15] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. "Hawk: Hybrid Datacenter Scheduling". In: *Proceedings of the USENIX Annual Technical Conference*. Santa Clara, California, USA, July 2015, pp. 499–510 (cited on pages 48, 51, 55, 59, 116, 166–167).

[Den68] Jack B. Dennis. "Programming generality, parallelism and computer architecture". In: 1968 (cited on page 30).

[DG08] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Communications of the ACM* 51.1 (Jan. 2008), pp. 107–113 (cited on pages 16, 28–29, 31, 35, 48, 51, 68).

[DK13] Christina Delimitrou and Christos Kozyrakis. "Paragon: QoS-aware Scheduling for Heterogeneous Datacenters". In: *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Houston, Texas, USA, Mar. 2013, pp. 77–88 (cited on pages 17, 46–47, 55, 135).

[DK14] Christina Delimitrou and Christos Kozyrakis. "Quasar: Resource-Efficient and QoS-Aware Cluster Management". In: *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Salt Lake City, Utah, USA, Mar. 2014 (cited on pages 17, 47, 51–52, 54–55, 135).

[DSK15] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. "Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters". In: *Proceedings of the 6th ACM Symposium on Cloud Computing (SoCC)*. Kohala Coast, Hawaii, USA, Aug. 2015, pp. 97–110 (cited on pages 17, 55, 57, 60, 114, 116, 152).

[EK72] Jack Edmonds and Richard M. Karp. "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems". In: *Journal of the ACM* 19.2 (Apr. 1972), pp. 248–264 (cited on page 129).

[FBK+12] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. "Jockey: Guaranteed Job Latency in Data Parallel Clusters". In: *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*. Bern, Switzerland, Apr. 2012, pp. 99–112 (cited on pages 52, 55).

[FF57] Lester R. Ford and Delber R. Fulkerson. "A primal-dual algorithm for the capacitated Hitchcock problem". In: *Naval Research Logistics Quarterly* 4.1 (1957), pp. 47–54 (cited on page 127).

[FLN16] Apache Software Foundation. *Apache Flink*. `http://flink.apache.org`; accessed 10/11/2016 (cited on pages 29, 32, 34).

[FM06] Antonio Frangioni and Antonio Manca. "A Computational Study of Cost Reoptimization for Min-Cost Flow Problems". In: *INFORMS Journal on Computing* 18.1 (2006), pp. 61–70 (cited on page 131).

[Ful61] Delbert R. Fulkerson. "An out-of-kilter method for minimal-cost flow problems". In: *Journal of the Society for Industrial and Applied Mathematics* 9.1 (1961), pp. 18–27 (cited on page 128).

[GGS+15] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, et al. "Broom: sweeping out Garbage Collection from Big Data systems". In: *Proceedings of the 15th USENIX/SIGOPS Workshop on Hot Topics in Operating Systems (HotOS)*. Kartause Ittingen, Switzerland, May 2015 (cited on page 21).

[GIA17] Ionel Gog, Michael Isard, and Martín Abadi. *Falkirk: Rollback Recovery for Dataflow Systems*. In submission. 2017 (cited on page 21).

[GIR16] Apache Software Foundation. *Apache Giraph*. `http://giraph.apache.org`; accessed 14/11/2016 (cited on pages 29, 32, 35–36).

[GJS76] M. R. Garey, D. S. Johnson, and L. Stockmeyer. "Some simplified NP-complete graph problems". In: *Theoretical Computer Science* 1.3 (1976), pp. 237–267 (cited on page 87).

[GK93] Andrew V. Goldberg and Michael Kharitonov. "On Implementing Scaling Push-Relabel Algorithms for the Minimum-Cost Flow Problem". In: *Network Flows and Matching: First DIMACS Implementation Challenge*. Edited by D.S. Johnson and C.C. McGeoch. DIMACS series in discrete mathematics and theoretical computer science. American Mathematical Society, 1993 (cited on page 129).

[GKR+16] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. "GRAPHENE: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters". In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Savannah, Georgia, USA, Nov. 2016, pp. 81–97 (cited on page 64).

[Gle15] Adam Gleave. "Fast and accurate cluster scheduling using flow networks". Computer Science Tripos Part II Dissertation. University of Cambridge Computer Laboratory, May 2015 (cited on page 19).

[GLG+12] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs". In: *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Hollywood, California, USA, Oct. 2012, pp. 17–30 (cited on pages 16, 28–29, 32, 35–37, 72).

[GOF16] Jeff Dean. *Software Engineering Advice from Building Large-Scale Distributed Systems*. `http://static.googleusercontent.com/media/research.google.com/en//people/jeff/stanford-295-talk.pdf`; accessed 13/11/2016 (cited on page 30).

[Gol97] Andrew V. Goldberg. "An Efficient Implementation of a Scaling Minimum-Cost Flow Algorithm". In: *Journal of Algorithms* 22.1 (1997), pp. 1–29 (cited on pages 119, 129, 143).

[GSC+15] Ionel Gog, Malte Schwarzkopf, Natacha Crooks, Matthew P. Grosvenor, Allen Clement, and Steven Hand. "Musketeer: all for one, one for all in data processing systems". In: *Proceedings of the 10th ACM European Conference on Computer Systems (EuroSys)*. Bordeaux, France, Apr. 2015 (cited on page 20).

[GSG+15] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. "Queues don't matter when you can JUMP them!" In: *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Oakland, California, USA, May 2015 (cited on pages 20, 36, 49, 156).

[GSG+16] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N.M̃. Watson, and Steven Hand. "Firmament: fast, centralized cluster scheduling at scale". In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Savannah, Georgia, USA, 2016, pp. 99–115 (cited on pages 20, 54).

[GSW15] Andrey Goder, Alexey Spiridonov, and Yin Wang. "Bistro: Scheduling Data-Parallel Jobs Against Live Production Systems". In: *Proceedings of the USENIX Annual Technical Conference*. Santa Clara, California, USA, July 2015, pp. 459–471 (cited on pages 25, 54–55, 57).

[GT89] Andrew V. Goldberg and Robert E. Tarjan. "Finding Minimum-cost Circulations by Canceling Negative Cycles". In: *Journal of the ACM* 36.4 (Oct. 1989), pp. 873–886 (cited on page 127).

[GT90] Andrew V. Goldberg and Robert E. Tarjan. "Finding Minimum-Cost Circulations by Successive Approximation". In: *Mathematics of Operations Research* 15.3 (Aug. 1990), pp. 430–466 (cited on page 129).

[GTT90] Andrew V. Goldberg, Éva Tardos, and Robert E. Tarjan. "Network flow algorithms". In: *Flows, Paths and VLSI Layout* (1990), pp. 101–164 (cited on page 130).

[GXD+14] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. "GraphX: Graph Processing in a Distributed Dataflow Framework". In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, Colorado, USA, Oct. 2014, pp. 599–613 (cited on pages 29, 39, 68, 77, 82).

[GZH+11] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. "Dominant resource fairness: fair allocation of multiple resource types". In: *Proceedings of the 8th USENIX Symposium on Networked System Design and Implementation (NSDI)*. Boston, Massachusetts, USA, Mar. 2011 (cited on pages 53, 155).

[GZS+13] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. "Choosy: max-min fair sharing for datacenter jobs with constraints". In: *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, Apr. 2013, pp. 365–378 (cited on pages 54–55).

[HAD16] Apache Software Foundation. *Apache Hadoop*. `http://hadoop.apache.org/`; accessed 13/11/2016 (cited on pages 31–32, 42).

[HBB+12] Alexander Hall, Olaf Bachmann, Robert Büssow, Silviu Gănceanu, and Marc Nunkesser. "Processing a Trillion Cells Per Mouse Click". In: *Proceedings of the VLDB Endowment* 5.11 (July 2012), pp. 1436–1446 (cited on page 38).

[HCS+12] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. "Green-Marl: A DSL for Easy and Efficient Graph Analysis". In: *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. London, England, United Kingdom, 2012, pp. 349–362 (cited on page 72).

[HFS16] Apache Hadoop. *Hadoop Fair Scheduler*. Accessed 11/09/2016. URL: `http://hadoop.apache.org/common/docs/stable1/fair_scheduler.html` (cited on page 53).

[HIV16] Ashish Thusoo. *Hive - A Petabyte Scale Data Warehouse using Hadoop*. `https://www.facebook.com/notes/facebook-engineering/hive-a-petabyte-scale-data-warehouse-using-hadoop/89508453919/`; accessed 28/11/2016 (cited on page 69).

[HKZ+11] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, et al. "Mesos: A platform for fine-grained resource sharing in the data center". In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. Boston, Massachusetts, USA, Mar. 2011, pp. 295–308 (cited on pages 45, 49, 51, 55, 57).

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. "Dryad: Distributed Data-parallel Programs from Sequential Building Blocks". In: *Proceedings of the 2nd ACM SIGOPS European Conference on Computer Systems (EuroSys)*. Lisbon, Portugal, Mar. 2007, pp. 59–72 (cited on pages 29, 32–33, 76).

[IPC+09] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. "Quincy: fair scheduling for distributed computing clusters". In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*. Big Sky, Montana, USA, Oct. 2009, pp. 261–276 (cited on pages 18, 48–51, 53–57, 115, 117–120, 146–147, 150).

[IZ10] Ming-Yee Iu and Willy Zwaenepoel. "HadoopToSQL: A MapReduce Query Optimizer". In: *Proceedings of the 5th ACM European Conference on Computer Systems (EuroSys)*. Paris, France, 2010, pp. 251–264 (cited on pages 69, 91).

[JSB+15] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. "Silo: Predictable Message Latency in the Cloud". In: *Proceedings of the 2015 ACM SIGCOMM Conference (SIGCOMM)*. London, United Kingdom, 2015, pp. 435–448 (cited on page 53).

[KAA+13] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. "Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing". In: *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, 2013, pp. 169–182 (cited on page 76).

[KBF+15] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, et al. "Twitter Heron: Stream Processing at Scale". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Melbourne, Victoria, Australia, 2015, pp. 239–250 (cited on page 30).

[KBG12] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. "GraphChi: Large-Scale Graph Computation on Just a PC". In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Hollywood, California, USA, 2012, pp. 31–46 (cited on pages 28–29, 32, 35, 37, 72).

[KBM17] Cloud Native Computing Foundation. *Kubernetes Kubemark*. `https://github.com/kubernetes/community/blob/master/contributors/design-proposals/kubemark.md`; accessed 28/06/2017 (cited on page 161).

[KD98] Navin Kabra and David J. DeWitt. "Efficient Mid-query Re-optimization of Sub-optimal Query Execution Plans". In: *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Seattle, Washington, USA, 1998, pp. 106–117 (cited on page 67).

[KIY13] Qifa Ke, Michael Isard, and Yuan Yu. "Optimus: A Dynamic Rewriting Framework for Data-parallel Execution Plans". In: *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, 2013, pp. 15–28 (cited on pages 33, 67, 76–77).

[KK12] Zoltán Király and P. Kovács. "Efficient implementations of minimum-cost flow algorithms". In: *Acta Universitatis Sapientiae* 4.1 (2012), pp. 67–118 (cited on pages 131, 141).

[KL70] Brian W. Kernighan and Shen Lin. "An efficient heuristic procedure for partitioning graphs". In: *Bell System Technical Journal* 49.2 (1970), pp. 291–307 (cited on page 87).

[Kle67] Morton Klein. "A Primal Method for Minimal Cost Flows with Applications to the Assignment and Transportation Problems". In: *Management Science* 14.3 (1967), pp. 205–220 (cited on page 127).

[KPX+11] Qifa Ke, Vijayan Prabhakaran, Yinglian Xie, Yuan Yu, Jingyue Wu, and Junfeng Yang. "Optimizing Data Partitioning for Data-Parallel Computing." In: *Proceedings of the 13th USENIX Workshop on Hot Topics in Operating Systems (HotOS)*. Napa, California, USA, May 2011 (cited on page 70).

[KRC+15] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, et al. "Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters". In: *Proceedings of the USENIX Annual Technical Conference*. Santa Clara, California, USA, July 2015, pp. 485–497 (cited on pages 48, 51, 55, 59, 166–167).

[KUB16] Cloud Native Computing Foundation. *Kubernetes*. `http://k8s.io`; accessed 14/09/2016 (cited on pages 45, 51).

[LA04] Chris Lattner and Vikram Adve. "LLVM: a compilation framework for lifelong program analysis transformation". In: *International Symposium on Code Generation and Optimization (CGO)*. Mar. 2004, pp. 75–86 (cited on pages 67, 75, 77).

[LBG+12] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud". In: *Proceedings of the VLDB Endowment* 5.8 (Apr. 2012), pp. 716–727 (cited on page 36).

[LCG+15] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. "Heracles: Improving Resource Efficiency at Scale". In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*. Portland, Oregon, USA, June 2015, pp. 450–462 (cited on pages 47–48).

[LGZ+14] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. "Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks". In: *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*. Seattle, Washington, USA, 2014, 6:1–6:15 (cited on page 49).

[Liu12] Huan Liu. *Host Server CPU utilization in Amazon EC2 cloud*. `https://tinyurl.com/hn5yh9d`; accessed 14/11/2015. 2012 (cited on page 60).

[LND16] Derrek G. Murray. *Building new frameworks on Naiad*. Apr. 2014 (cited on pages 29, 39).

[Löb96] Andreas Löbel. *Solving Large-Scale Real-World Minimum-Cost Flow Problems by a Network Simplex Method*. Technical report SC-96-07. Zentrum für Informationstechnik Berlin (ZIB), Feb. 1996 (cited on page 131).

[MAA+14] Stefan C. Müller, Gustavo Alonso, Adam Amara, and André Csillaghy. "Pydron: Semi-Automatic Parallelization for Multi-Core and the Cloud". In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, Colorado, USA, Oct. 2014, pp. 645–659 (cited on page 67).

[MAB+10] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. "Pregel: A System for Large-scale Graph Processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Indianapolis, Indiana, USA, June 2010, pp. 135–146 (cited on pages 28–29, 32, 35–36, 72).

[MAH16] Apache Software Foundation. *Apache Mahout*. `http://mahout.apache.org/`; accessed 14/11/2016 (cited on page 34).

[MAP+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Whitepaper available at `http://tensorflow.org`. 2015 (cited on page 20).

[McK08] McKinsey & Company. "Revolutionizing data center efficiency". In: (2008) (cited on page 60).

[McS14] Frank McSherry. *GraphLINQ: A graph library for Naiad*. Big Data at SVC blog, accessed 25/07/2016. May 2014. URL: `http://bigdataatsvc.wordpress.com/2014/05/08/graphlinq-a-graph-library-for-naiad/` (cited on pages 29, 39, 68, 72).

[MG12] Raghotham Murthy and Rajat Goel. "Peregrine: Low-latency Queries on Hive Warehouse Data". In: *XRDS: Crossroad, ACM Magazine for Students* 19.1 (Sept. 2012), pp. 40–43 (cited on page 38).

[MGL+10] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. "Dremel: Interactive Analysis of Web-scale Datasets". In: *Proceedings of the VLDB Endowment* 3.1-2 (Sept. 2010), pp. 330–339 (cited on pages 28, 32).

[MIM15] Frank McSherry, Michael Isard, and Derek G. Murray. "Scalability! But at what COST?" In: *Proceedings of the 15th USENIX/SIGOPS Workshop on Hot Topics in Operating Systems (HotOS)*. Kartause Ittingen, Switzerland, May 2015 (cited on page 36).

[MMI+13] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. "Naiad: A Timely Dataflow System". In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. Nemacolin Woodlands, Pennsylvania, USA, Nov. 2013, pp. 439–455 (cited on pages 16, 28–29, 32, 68, 71, 156).

[MMK10] Yandong Mao, Robert Morris, and M. Frans Kaashoek. *Optimizing MapReduce for multicore architectures*. Technical report MIT-CSAIL-TR-2010-020. Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Laboratory, 2010 (cited on pages 29, 32).

[MSS+11] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. "CIEL: a universal execution engine for distributed data-flow computing". In: *Proceedings of the 8th USENIX Symposium on Networked System Design and Implementation (NSDI)*. Boston, Massachusetts, USA, Mar. 2011, pp. 113–126 (cited on pages 29, 32, 34, 67).

[MT13] Jason Mars and Lingjia Tang. "Whare-map: Heterogeneity in "Homogeneous" Warehouse-scale Computers". In: *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*. Tel-Aviv, Israel, June 2013, pp. 619–630 (cited on pages 46, 55).

[MTH+11] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations". In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Porto Allegre, Brazil, Dec. 2011, pp. 248–259 (cited on page 47).

[Mur11] Derek G. Murray. "A distributed execution engine supporting data-dependent control flow". PhD thesis. University of Cambridge Computer Laboratory, July 2011 (cited on page 76).

[NEF+12] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. "Flat Datacenter Storage". In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Hollywood, California, USA, Oct. 2012, pp. 1–15 (cited on page 148).

[NIG07] Ripal Nathuji, Canturk Isci, and Eugene Gorbatov. "Exploiting platform heterogeneity for power efficient data centers". In: *Proceedings of the 2007 International Conference on Autonomic Computing (ICAC)*. Jacksonville, Florida, USA, 2007 (cited on page 46).

[NRN+10] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. "S4: Distributed Stream Computing Platform". In: *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops (ICDMW)*. Sydney, Australia, Dec. 2010, pp. 170–177 (cited on page 30).

[OOZ16] Apache Software Foundation. *Apache Oozie*. `http://oozie.apache.org/`; accessed 14/11/2016 (cited on pages 33, 44).

[OPR+13] Kay Ousterhout, Aurojit Panda, Joshua Rosen, Shivaram Venkataraman, Reynold Xin, Sylvia Ratnasamy, Scott Shenker, et al. "The case for tiny tasks in compute clusters". In: *Proceedings of the 14th USENIX Workshop on Hot Topics in Operating Systems (HotOS)*. Santa Ana Pueblo, New Mexico, USA, May 2013 (cited on pages 42, 167).

[Orl93] James B. Orlin. "A faster strongly polynomial minimum cost flow algorithm". In: *Operations research* 41.2 (1993), pp. 338–350 (cited on page 130).

[ORS+08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. "Pig Latin: A Not-so-foreign Language for Data Processing". In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Vancouver, Canada, 2008, pp. 1099–1110 (cited on pages 29, 33, 38, 44, 68–70, 77).

[ORS+11] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. "Fast Crash Recovery in RAMCloud". In: *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*. Cascais, Portugal, Oct. 2011, pp. 29–41 (cited on page 49).

[OWZ+13] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. "Sparrow: Distributed, Low Latency Scheduling". In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. Nemacolin Woodlands, Pennsylvania, USA, Nov. 2013, pp. 69–84 (cited on pages 17, 48, 50–51, 54–57, 59, 114, 116, 154, 165–166).

[OZN+12] Zhonghong Ou, Hao Zhuang, Jukka K. Nurminen, Antti Ylä-Jääski, and Pan Hui. "Exploiting Hardware Heterogeneity Within the Same Instance Type of Amazon EC2". In: *Proceedings of the 4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*. Boston, Massachusetts, USA, June 2012 (cited on page 46).

[PDG+05] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. "Interpreting the data: Parallel analysis with Sawzall". In: *Scientific Programming* 13.4 (2005), pp. 277–298 (cited on pages 29, 38).

[PE95] Bill Pottenger and Rudolf Eigenmann. "Idiom Recognition in the Polaris Parallelizing Compiler". In: *Proceedings of the 9th International Conference on Supercomputing (ICS)*. Barcelona, Spain, 1995, pp. 444–448 (cited on page 82).

[PKC+12] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. "FairCloud: Sharing the Network in Cloud Computing". In: *Proceedings of the 2012 ACM SIGCOMM Conference (SIGCOMM)*. Helsinki, Finland, 2012, pp. 187–198 (cited on page 53).

[Pla13] David A Plaisted. "Source-to-Source Translation and Software Engineering". In: *Software Engineering and Applications* 6.suppl 4A (2013), p. 30 (cited on pages 72, 79).

[PTS+17] Shoumik Palkar, James J. Thomas, Anil Shanbhag, Malte Schwarzkopf, Saman Amarasinghe, and Matei Zaharia. "A Common Runtime for High Performance Data Analysis". In: *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research (CIDR)*. Chaminade, California, USA, Jan. 2017 (cited on page 75).

[RKK+16] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. "Efficient Queue Management for Cluster Scheduling". In: *Proceedings of the 11th ACM European Conference on Computer Systems (EuroSys)*. London, United Kingdom, 2016, 36:1–36:15 (cited on pages 17, 55, 59, 114, 116, 132).

[RMZ13] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. "X-Stream: Edge-centric Graph Processing Using Streaming Partitions". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*. Farminton, Pennsylvania, USA, 2013, pp. 472–488 (cited on pages 29, 32, 37).

[RTG+12] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. "Heterogeneity and dynamicity of clouds at scale: Google trace analysis". In: *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC)*. San Jose, California, Oct. 2012, 7:1–7:13 (cited on pages 46, 48, 50–52, 54, 119, 161).

[SAM16] Apache Software Foundation. *Apache Samza*. `http://samza.apache.org`; accessed 13/11/2016 (cited on page 30).

[SCH+11] Bikash Sharma, Victor Chudnovsky, Joseph L. Hellerstein, Rasekh Rifaat, and Chita R. Das. "Modeling and synthesizing task placement constraints in Google compute clusters". In: *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC)*. Cascais, Portugal, Oct. 2011, 3:1–3:14 (cited on pages 50–51, 150).

[Sch16] Malte Schwarzkopf. "Operating system support for warehouse-scale computing". PhD thesis. University of Cambridge Computer Laboratory, Feb. 2016 (cited on pages 46–47, 115, 120–121, 145–148, 150, 152).

[SKA+13] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. "Omega: flexible, scalable schedulers for large compute clusters". In: *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, Apr. 2013, pp. 351–364 (cited on pages 17, 24, 54–55, 58, 114, 161).

[SPW09] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. "DRAM Errors in the Wild: A Large-Scale Field Study". In: *Proceedings of the 11th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. Seattle, WA, USA, 2009, pp. 193–204 (cited on page 30).

[SWC+12] Alkis Simitsis, Kevin Wilkinson, Malu Castellanos, and Umeshwar Dayal. "Optimizing Analytic Data Flows for Multiple Execution Engines". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Scottsdale, Arizona, USA, 2012, pp. 829–840 (cited on page 44).

[TCG+12] Alexey Tumanov, James Cipar, Gregory R. Ganger, and Michael A. Kozuch. "Alsched: Algebraic Scheduling of Mixed Workloads in Heterogeneous Clouds". In: *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC)*. San Jose, California, Oct. 2012, 25:1–25:7 (cited on pages 51, 55, 57, 150).

[TMV+11] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. "The impact of memory subsystem resource sharing on datacenter applications". In: *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*. San Jose, California, USA, June 2011, pp. 283–294 (cited on page 46).

[TSJ+09] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, et al. "Hive: A Warehousing Solution over a Map-reduce Framework". In: *Proceedings of the VLDB Endowment* 2.2 (Aug. 2009), pp. 1626–1629 (cited on pages 16, 29, 33, 38, 68–69, 77).

[TTS+14] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, et al. "Storm @Twitter". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Snowbird, Utah, USA, 2014, pp. 147–156 (cited on page 30).

[TZP+16] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. "TetriSched: Global Rescheduling with Adaptive Plan-ahead in Dynamic Heterogeneous Clusters". In: *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*. London, England, United Kingdom, 2016, 35:1–35:16 (cited on pages 51, 55, 57, 135, 150, 152).

[Val90] Leslie G. Valiant. "A Bridging Model for Parallel Computation". In: *Communications of the ACM* 33.8 (Aug. 1990), pp. 103–111 (cited on page 35).

[VMD+13] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, et al. "Apache Hadoop YARN: Yet Another Resource Negotiator". In: *Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC)*. Santa Clara, California, Oct. 2013, 5:1–5:16 (cited on pages 45, 48, 50–51, 55, 57).

[VPA+14] Shivaram Venkataraman, Aurojit Panda, Ganesh Ananthanarayanan, Michael J. Franklin, and Ion Stoica. "The Power of Choice in Data-Aware Cluster Scheduling". In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, Colorado, USA, Oct. 2014, pp. 301–316 (cited on page 55).

[VPK+15] Abhishek Verma, Luis David Pedrosa, Madhukar Korupolu, David Oppenheimer, and John Wilkes. "Large scale cluster management at Google". In: *Proceedings of the 10th ACM European Conference on Computer Systems (EuroSys)*. Bordeaux, France, Apr. 2015 (cited on pages 17, 24, 45, 50–52, 54, 56, 116, 161, 166).

[XRZ+13] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Shark: SQL and Rich Analytics at Scale". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. New York, New York, USA, 2013, pp. 13–24 (cited on pages 38, 68, 77).

[YIF+08] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language". In: *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. San Diego, California, USA, Dec. 2008 (cited on pages 29, 34, 39, 68, 71).

[ZBS+10] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. "Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling". In: *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*. Paris, France, Apr. 2010, pp. 265–278 (cited on pages 49, 55, 93).

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, et al. "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing". In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. San Jose, California, USA, Apr. 2012, pp. 15–28 (cited on pages 16, 28–29, 32, 34, 42, 49, 68, 76, 156).

[ZKJ+08] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy H. Katz, and Ion Stoica. "Improving MapReduce Performance in Heterogeneous Environments". In: *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. San Diego, California, USA, Dec. 2008, pp. 29–42 (cited on pages 49, 55).

[ZTH+13] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. "CPI^2: CPU Performance Isolation for Shared Compute Clusters". In: *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, Apr. 2013, pp. 379–391 (cited on page 54).

[ZWC+16] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. "Exploring the Hidden Dimension in Graph Processing". In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Savannah, Georgia, USA, Nov. 2016, pp. 285–300 (cited on page 37).