Resource Management for Advanced Data Analytics at Large Scale

Haoyu Zhang

A Dissertation

Presented to the Faculty

of Princeton University

in Candidacy for the Degree

of Doctor of Philosophy

Recommended for Acceptance

by the Department of

Computer Science

Adviser: Professor Michael J. Freedman

June 2018

Abstract

The rapidly growing size of data and the complexity of analytics present new challenges for large-scale data analytics systems. Modern distributed computing frameworks need to support not only embarrassingly parallelizable batch jobs, but also advanced applications analyzing text and multimedia data using complex queries and machine learning (ML) models. Given the computation and storage costs of advanced data analytics, resource management is crucial. New applications and workloads expose vastly different characteristics which make traditional scheduling systems inadequate, and at the same time offer great opportunities that lead to new system designs for better performance.

In this thesis, we present resource management systems that significantly improve cloud resource efficiency by leveraging the specific characteristics of advanced data analytics applications. We present the design and implementation of the following systems:

(i) VideoStorm: a video analytics system that scales to process thousands of vision queries on live video streams over large clusters. VideoStorm's offline profiler generates resource-quality profiles for vision queries, and its online scheduler allocates resources to maximize performance in terms of vision processing quality and lag.

(ii) SLAQ: a cluster scheduling system for approximate ML training jobs that aims to maximize the overall model quality. In iterative and exploratory training settings, better models can be obtained faster by directing resources to jobs with the most potential for improvement. SLAQ allocates resources to maximize the cluster-wide quality improvement based on highly-tailored model quality predictions.

(iii) Riffle: an optimized shuffle service for big-data analytics frameworks that significantly improves I/O efficiency. The all-to-all data transfer (i.e., shuffle) in modern big-data systems (such as Spark and Hadoop) becomes the scaling bottleneck for multi-stage analytics jobs, due to the superlinear increase in disk I/O operations as data volume grows. Riffle boosts system performance by merging fragmented intermediate files and efficiently scheduling the merge operations.

Taken together, this thesis demonstrates a novel set of methods in both job-level and task-level scheduling for building scalable, highly-efficient, and cost-effective resource management systems. We have performed extensive evaluation with real production workloads, and our results show significant improvement in resource efficiency, job completion time, and system throughput for advanced data analytics.

Acknowledgments

I am deeply grateful to my advisor, Mike Freedman, for his guidance, support, and encouragement in the past five years. Working with Mike is always productive and inspiring. His pursuit of high standards in research greatly motivated me to conduct impactful work and make intellectual contributions in computer systems. Mike has always been patient and encouraging, and opened for me the gates of scientific research in distributed systems and various other areas. My Ph.D. life would not have been so rewarding and enjoyable without my advisor.

I feel extremely fortunate to have Kyle Jamieson, Kai Li, Wyatt Lloyd, and Jennifer Rexford as my thesis committee members. Jen has been an awesome mentor who gave constructive suggestions and kind support on both research and graduate life. Kai, Kyle, and Wyatt provided their insightful advice and benevolent help with my research work at different stages of my Ph.D. study. I am also thankful to Nick Feamster for being my general exam committee member, and Aarti Gupta for her advice on research projects. I have experienced an engaging and collaborative academic environment in the systems and networking lab thanks to these amazing professors.

During my internship at Microsoft Research, I received tremendous support from my mentors Ganash Ananthanarayanan and Peter Bodik. Ganesh is always sharp with his questions, and always persistent in overcoming technical challenges. Peter is an impressive system designer and an experienced system builder. Their vision and passion profoundly influenced me in every aspect of being a good researcher.

I benefit immensely from my internship at the Facebook BigCompute group. My mentors Brian Cho and Ergin Seyfe guided me through every step of engineering a production-quality system. They demonstrated to me how to build scalable and reliable systems with real-world impact. Brian also helped me dedicatedly during the whole process of formalizing the Riffle research project and writing the paper.

I sincerely appreciate the professional and personal friendship with my outstanding collaborators at Princeton. I am grateful to Xiaozhou and Xin for the diligent work and thought-provoking discussions during our collaboration on research projects. I am indebted to Naga who worked closely with me on the first research project at Princeton. I thank Logan and Andrew for their hard work and contribution to the SLAQ project. I would also like to thank my collaborators and coauthors from Microsoft Research, Facebook, Barefoot Networks, Cornell, and UC Berkeley.

I had a great time working together with the SNS group current and former members, and I would like to give special thanks to Marcela, Daniel, Amy, Sid, and Aaron for reading and helping me improve paper drafts. I had so much joy and fun during the time I spent with my friends. Beyond those already mentioned above, thanks to Linpeng, Yihan, Zhen, Yushan, Zhenyu, Zhixing, Xinyi, Nanxi, Yinda, Yichen, Zeyu, Annie, Haipeng, Li, Kelvin, Zhuqi, Qian, Wei, Haowen, Chengwei, Tongxin, Yapeng, Yitong, Xiaoyu, Xueying, Shiyu, Lin, Yunxi, Tianyi, and Dannie. Thank you all for making this journey enjoyable.

This dissertation work is supported by National Science Foundation (NSF) Awards CNS-0953197 (CAREER) and IIS-1250990 (BIGDATA).

Finally, I wish to thank my parents for their unconditional love and support. They have always believed in me and gave me the strength to complete what I started.

To my parents.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Advanced Data Analytics Systems

Advanced data analytics is a broad category of queries that can help human discover interpretable patterns, understand causal relationships, and make informed decisions in practical problems by analyzing a large amount of data. Compared to simple data analytics which only involves easily parallelizable operators on partitions of data, advanced data analytics queries have broader capability and better expressiveness. These queries are becoming increasingly important to gain deep insights and drive operational improvements based on sophisticated processing logic on massive amounts of data, which in turn present new efficiency, scalability, and reliability challenges to underlying systems.

From structured data to multimodal data. The development of data acquisition, storage, and retrieval techniques brings a revolutionary transformation to data analytics systems. The focus of big-data research has expanded from structured or semi-structured data of text and numerical types, to unstructured *multimodal* data such as still images, audios, videos, and interconnected corpus with links and click behaviors. Video processing, among others, has a wide variety of commercial and security applications. Security cameras in buildings are deployed for surveillance and business intelligence (i.e., identifying people and their

actions), while cameras deployed at street intersections are used for traffic control (e.g., counting car volumes) and crime prevention.

Computer vision researchers and developers offer a wide variety of vision analytics modules focusing on different types of video processing tasks. Modules exist for purposes of security (e.g., motion detection, loitering, unattended luggage), traffic management (license plate recognition, parking violations), city planning (counting cars, bicycles, jay walking), or retail management (recognizing and monitoring customers in a store). These applications are drawing a growing interest from both academic research and commercial companies to design new systems [102, 116, 195, 198].

From handcrafted expert systems to machine learning models. Handcrafted expert systems encode enormous human knowledge to handle particular problems by mimicking the way that human experts think and solve them [57]. However, constructing such models typically requires strong expertise in domain specific knowledge and mathematics, and difficult to evolve as the accuracy requirement and model complexity continue to grow. Machine learning (ML) becomes an increasingly important tool for large-scale data analytics; it has been successfully deployed for online search, marketing, healthcare, machine translation, and information security.

Compared to expert systems, ML training is computationally expensive and usually requires multiple passes over the entire datasets. The model performance depends on the entire training stack including model structures, optimization algorithms, software architecture of the training frameworks, and computing hardwares. It is challenging to efficiently manage system resources and facilitate time-sensitive ML training on large datasets.

From simple queries to multi-stage computation jobs. Large-scale batch analytics queries are prevalently used in large companies holding and constantly generating big data. Distributed data analytics engines, such as Spark [191], MapReduce [81], and Dryad [105],

are widely used for executing SQL queries and user-defined functions (UDFs), performing graph analytics [89], and preprocessing and postprocessing datasets in ML jobs.

Based on the business requirements, data processing pipelines are often expressed as a *combination* of multiple relational queries and complex procedural algorithms [51]. Accordingly, the big-data frameworks need to perform multiple stages of filtering, transformation, joining, and repartitioning the datasets during the job execution. Multi-stage operations trigger the datasets to be frequently exchanged between distributed machines, which incur huge network and storage I/O overhead and, without close attention, significantly impair the scalability and performance.

1.2 Challenges for Existing Big-Data Platforms

The challenge in analyzing massive data using advanced queries arises from the fact that the volume and complexity of data processing grow much faster than hardware speed and capacity improvements. We focus on the data and complexity challenge in the context of limited hardware resources in clusters.

The data challenge. Video cameras constantly generate large volume of data, and sometimes require the data to be processed almost in real time. It has been reported that major cities all around the world have millions of cameras deployed [55, 99].

Datasets used for training ML models are also growing. The Netflix movie rating dataset [28] includes more than 100 million user ratings of 17,000 movies, and the ImageNet dataset [82] includes over 14 million images of thousands of categories. In the *deep learning* era, we are likely to see more of such datasets to become available.

As for batch processing, the Spark deployment at Facebook processes tens of PBs of data per day. One single job processing a key dataset analyzes hundreds of TBs of newly generated data per day. The data volume is typically more than 10 times larger than the available memory, which makes it impossible to cache the entire dataset inside memory.

The complexity challenge. Video analytics, due to highly complicated processing logic, can have extremely high resource demands. Tracking objects in video is a core primitive for many scenarios, but the best tracker [145] in the VOT Challenge 2015 [121] processes only 1 frame per second on an 8-core machine. Some of the most accurate Deep Neural Networks for object recognition, another core primitive, require 30GFlops to process a single frame per second [165].

For ML workloads, a training job typically takes more than one thousand steps for the model to converge. Moreover, ML training is not a one-time effort. ML practitioners need to repeatedly train the same model to explore and find the best feature set, hyperparameter configurations, and model structures; it has been reported that people need to explore up to thousands of hyperparameter combinations to get the best model [134].

Executing multi-stage batch processing queries also involve high resource demand. For example, more than 50% of the production jobs at Facebook are multi-stage jobs that require multiple passes on the data. While one can tune the resource allocation and execution plan to get the best performance of a job, this solution is untenable for thousands of jobs at Facebook. Each job has different characteristics (e.g., distribution and skew of data) which also change over time depending on outside factors such as Facebook user behavior, so it is not possible to find the optimal plans without tedious experimentation.

Limited cluster resources. Despite the growing resource demands for advanced data analytics, the underlying hardware and cloud resources, unfortunately, can no longer get fast or cheap quickly enough after the end of Moore's Law [173]. The slowdown in the increase of integration density and clock frequency makes it more difficult to meet the various requirements of data analytics applications—they cannot reckon that performance gains come effortlessly from new-generation hardware. Advanced data analytics jobs at large scale urge the efficient allocation of cloud resources for performance and scalability.

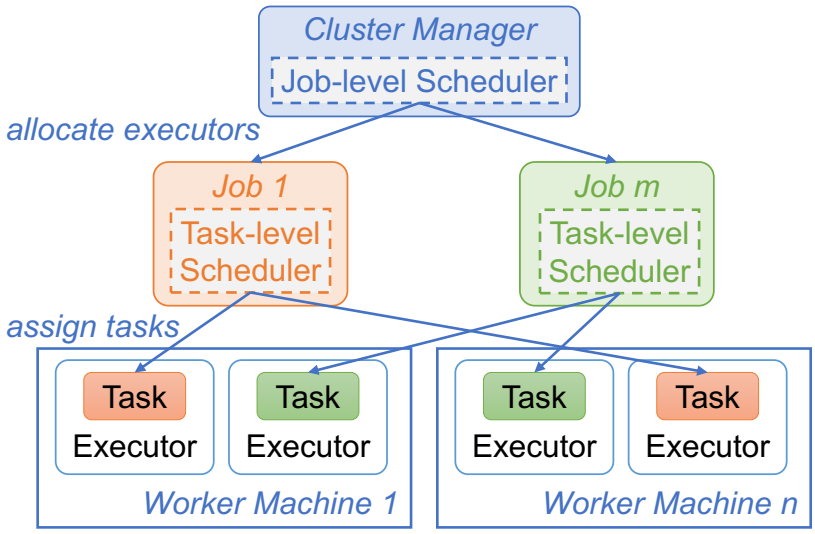


Figure 1.1: Hierarchical resource management system.

1.3 Overview of Resource Management

Resource management systems are widely used for the coordination and execution of concurrent job contending for shared resources in datacenters. Each cluster machine is typically sliced into smaller executors (virtual machines or containers) for efficient resource multiplexing (i.e., hosting tasks from multiple concurrent jobs) and failure isolation. Resources of distributed computing frameworks are typically managed by hierarchical schedulers [19, 20, 59, 103].

In general, the resource manager can be organized in a two-layer architecture, as shown in Figure 1.1. The *job-level scheduler* is in charge of allocating resources between multiple concurrent jobs, For example, fair schedulers [3, 59, 65, 79, 87, 91, 100, 181] are widely used as job-level resource managers for big-data frameworks. They take resource demands from individual jobs, and mostly allocate resources based on fairness. Extending the functionality of fair allocation, cluster schedulers nowadays also support a rich set of scheduling policies such as reservation, deadline, and priority [79, 84, 114].

The *task-level scheduler*, on the other hand, decides how to assign tasks on multiple task executors for each job. Task is the basic executing unit to schedule and launch inside a job.

For example, a task in the context of batch processing is an instance of several operations processing one partition of the dataset; task in stream processing is commonly dealing with data items arrived in a specific time window. Key to designing an efficient task-level scheduler for big-data frameworks is to generate an intelligent execution plan of dividing a job into tasks, and to dispatch a large amount of tasks on multiple executors.

With the data and complexity challenges presented by advanced data analytics, resource management is crucial. We observe that it is essential for the cluster schedulers to understand the specific characteristics and core requirements from various advanced applications in order to achieve better efficiency and cost-effectiveness of resources, as well as high performance of applications.

Traditional cluster schedulers treat individual tasks as black boxes, and thus schedule resources according to general policies and strategies, including fairness (max-min fairness, dominant resource fairness), priorities, and deadlines. Being agnostic to workload specific characteristics, the general-purpose job schedulers fall short to meet various requirements of these applications, and the general-purpose task schedulers cannot achieve the most efficient resource allocation when executing jobs.

In fact, advanced data analytics expose new opportunities to optimize scheduling decisions. Our key insight is that *by leveraging application-specific features, we can optimize resource scheduling decisions and achieve better performance for advanced data analytics*.

In this thesis, we present the design of resource management systems for three scenarios in advanced data analytics applications. The deployment of these prevalent big-data processing applications in large companies typically takes up an entire cluster or even spans multiple clusters. Thus the specialization of scheduling systems could lead to significant improvement in not only the system performance, but also the overall resource efficiency.

- We observe that vision analytics workloads expose a unique quality-delay-resource trade-off, and our system, VideoStorm, takes advantage of this trade-off to jointly optimize processing quality and lag with multiple concurrent queries.

- ML training is typically an iterative process with diminishing returns. We designed SLAQ, a quality-driven scheduler to optimize the cluster-wide performance of distributed ML jobs on shared resources.
- We identify that the scaling bottleneck of multi-stage batch processing jobs is the fragmented I/O requests during shuffle operations. We built Riffle to optimize the shuffle service for efficient task execution that scales to process petabytes of data.

1.4 Contributions

VideoStorm [195] is a video analytics system that scales to processing thousands of *live* video streams over large clusters. The system leverages important features of video analytics queries, namely the resource-quality trade-off with multi-dimensional configurations, and the variety in quality and lag goals. At its core, VideoStorm contains an offline profiler that efficiently generates the query's *resource-quality profile* for its different configurations, and an online scheduler *jointly maximizes the quality and minimizes the lag* of streaming video queries. In doing so, it uses the generated profiles, and lag and quality goals. It allocates resources to each query and picks its configuration based on the allocation. We evaluated VideoStorm using real video analytics queries over video datasets from operational traffic cameras from cities we partnered with. The VideoStorm scheduler outperforms fair scheduling of resources by as much as 80% in quality of queries and 7× in terms of lag.

SLAQ [197] is a cluster scheduling system for ML training jobs that aims to maximize the overall job quality. SLAQ dynamically allocates resources based on job resource demands, intermediate model quality, and the system's workload. The intuition behind SLAQ is that in the context of approximate ML training, more resources should be allocated to jobs that have the most potential for quality improvement. SLAQ leverages the fact that most ML training algorithms are implemented as an iterative optimization process. By continually

monitoring the history of quality improvement and runtime, SLAQ generates highly-tailored and accurate quality predictions for future training iterations. SLAQ estimates the impact of resource allocation on model quality, and explores the quality-runtime trade-offs across multiple jobs. Based on this information, SLAQ adjusts their resource allocations of all running jobs to best utilize the limited cluster resources. The SLAQ scheduler is designed to be dynamic and fine-grained, so that resource allocations can adapt quickly to jobs' quality and the system's workload changes. We evaluate various distinct ML training algorithms on datasets collected from various online sources. We found that SLAQ improves the average quality by up to 73% and reduces the average delay by up to 44% compared to fair resource scheduling.

Riffle [196] is an optimized shuffle service for big-data analytics frameworks that significantly improves I/O efficiency and scales to processing PB-level data. Riffle boosts shuffle performance and improves resource efficiency by converting large amounts of small, random shuffle I/O requests into much fewer large, sequential I/O requests. At its core, Riffle consists of a *centralized scheduler* that keeps track of intermediate shuffle files and dynamically coordinates merge operations, and a *shuffle merge service* which runs on each physical cluster node and efficiently merges the small files into larger ones with little resource overhead. Riffle has to be efficient in handling shuffle files without using much computation or storage resources, is easy to configure to best fit different storage systems and hardware, and is robust and efficient when handling merge operation failures. We run experiments of Riffle on a representative mix of Facebook's production jobs processing 100s of TB of data: Riffle reduces disk I/O requests by up to 10x and the end-to-end job completion time by up to 40%.

Practical deployment of systems. We further demonstrate the applicability and reliability of our approaches by the experience of deploying them in real-world systems. VideoStorm is used as the core scheduling module for Microsoft's live video analytics platform, which is

currently deployed and running in the traffic departments of Bellevue, WA and Cambridge, U.K. processing live streams from thousands of operational traffic cameras. The Riffle-enabled Spark is fully deployed in the disaggregated clusters at Facebook. It is constantly processing data analytics jobs daily on tens of petabytes of newly generated data per day. Our experience running Riffle shows significant performance improvement on production jobs in Facebook's datacenters with hundreds of physical machines.

Chapter 2

VideoStorm: Live Video Analytics with Approximation and Delay Tolerance

Video cameras are pervasive; major cities worldwide like New York City, London, and Beijing have millions of cameras deployed [55, 99]. Cameras are installed in buildings for surveillance and business intelligence, while those deployed on streets are for traffic control and crime prevention. Key to achieving the potential of these cameras is effectively analyzing the *live* video streams.

Organizations that deploy these cameras—cities or police departments—operate large clusters to analyze the video streams [12, 17]. Sufficient bandwidth is provisioned (fiber drops or cellular) between the cameras and the cluster to ingest video streams. Some analytics need to run for long periods (e.g., counting cars to control traffic light durations) while others for short bursts of time (e.g., reading the license plates for AMBER Alerts[1]).

Video analytics can have *very high resource demands*. Tracking objects in video is a core primitive for many scenarios, but the best tracker [145] in the VOT Challenge 2015 [121] processes only 1 frame per second on an 8-core machine. Some of the most accurate Deep Neural Networks for object recognition, another core primitive, require 30GFlops to process

[1] AMBER Alerts are raised in U.S. cities to identify child abductors

a single frame [165]. Due to the high processing costs and high data-rates of video streams, resource management of *video analytics queries* is crucial. We highlight two properties of video analytics queries relevant to resource management.

Resource-quality trade-off with multi-dimensional configurations. Vision algorithms typically contain various parameters, or *knobs*. Examples of knobs are video resolution, frame rate, and internal algorithmic parameters, such as the size of the sliding window to search for objects in object detectors. A combination of the knob values is a query *configuration*. The configuration space grows exponentially with the number of knobs. *Resource demand* can be reduced by changing configurations (e.g., changing the resolution and sliding window size) but they typically also lower the output *quality*.

Variety in quality and lag goals. While many queries require producing results in real-time, others can tolerate *lag* of even many minutes. This allows for temporarily reallocating some resources from the lag-tolerant queries during interim shortage of resources. Such a shortage happens due to a burst of new video queries or "spikes" in resource usage of existing queries (for example, due to an increase in number of cars to track on the road).

Indeed, video analytics queries have a wide variety of quality and lag goals. A query counting cars to control the traffic lights can work with moderate quality (approximate car counts) but will need them with low lag. License plate readers at toll routes [36, 38], on the other hand, require high quality (accuracy) but can tolerate lag of even many minutes because the billing can be delayed. However, license plate readers when used for AMBER Alerts require high quality results *without* lag.

Scheduling large numbers of streaming video queries with diverse quality and lag goals, each with many configurations, is computationally complex. Production systems for stream processing like Storm [9], StreamScope [131], Flink [1], Trill [71], and Spark Streaming [192] allocate resources among multiple queries only based on *resource fairness* [19, 20, 59, 88, 108] common to cluster managers like Yarn [3] and Mesos [100]. While

simple, being agnostic to the quality and lag of queries makes fair sharing far from ideal for video stream analytics.

We present VideoStorm, a video analytics system that scales to processing thousands of *live* video streams over large clusters. Users submit video analytics queries containing many *transforms* that perform vision signal processing on the frames of the incoming video. At its core, VideoStorm contains a scheduler that efficiently generates the query's *resource-quality profile* for its different knob configurations, and then *jointly maximizes the quality and minimizes the lag* of streaming video queries. In doing so, it uses the generated profiles, and lag and quality goals. It allocates resources to each query and picks its configuration (knob values) based on the allocation.

Challenges and Solution. The major technical challenges for designing VideoStorm can be summarized as follows: *(i)* There are no analytical models for resource demand and quality for a query configuration, and the large number of configurations makes it expensive to even estimate the resource-quality profile. *(ii)* Expressing quality and lag goals of *individual queries* and *across all queries* in a cluster is non-trivial. *(iii)* Deciding allocations and configurations is a computationally hard problem *exponential* in the number of queries and knobs.

To deal with the multitude of knobs in video queries, we split our solution into *offline* (or profiling) and *online* phases. In the offline phase, we use an efficient *profiler* to get the resource-quality profile of queries without exploring the entire combinatorial space of configurations. Using greedy search and domain-specific sampling, we identify a handful of knob configurations on the *Pareto boundary* of the profile. The scheduler in the online phase, thus, has to consider only these configurations.

We encode quality and lag goals of a query in a *utility function*. Utility is a weighted combination of the achieved quality and lag, with penalties for violating the goals. Penalties allow for expressing priorities between queries. Given utilities of multiple queries, we

schedule for two natural objectives—maximize the *minimum* utility, or maximize the *total* utility. The former achieves fairness (max-min) while the latter targets performance.

Finally, in the online phase, we model the scheduling problem using the Model-Predictive Control [142] to predict the *future* query lag over a short time horizon, and use this predicted lag in the utility function. The scheduler considers the resource-quality profile of queries during allocation, and allows for lagging queries to "catch up." It also deals with inevitable inaccuracies in resource usages in the resource-quality profiles.

While we focus VideoStorm on video analytics using computer vision algorithms, approximation and lag are aspects that are fundamental to all machine learning algorithms. To that end, the techniques in our system are broadly applicable to all stream analytics systems that employ machine learning techniques.

Contributions. Our contributions are as follows:

1. We designed and built a system for large-scale analytics of live video that allows users to submit queries with arbitrary vision processors.
2. We efficiently identify the resource-quality profile of video queries without exhaustively exploring the combinatorial space of knob configurations.
3. We designed an efficient scheduler for video queries that considers their resource-quality profile and lag tolerance, and trades off between them.

We considered streaming databases with approximation [42, 73, 144] as a starting point for our solution. However, they only consider the sampling rate of data streams and used established analytical models [77] to calculate the quality and resource demand. In contrast, vision queries are more complex black-boxes with many more knobs, and do not have known analytical models. Moreover, they optimize only one query at a time, while our focus is on scheduling multiple concurrent queries.

Deployment on 101 machines in Azure show that VideoStorm's scheduler allocates resources in hundreds of milliseconds even with thousands of queries. We evaluated using

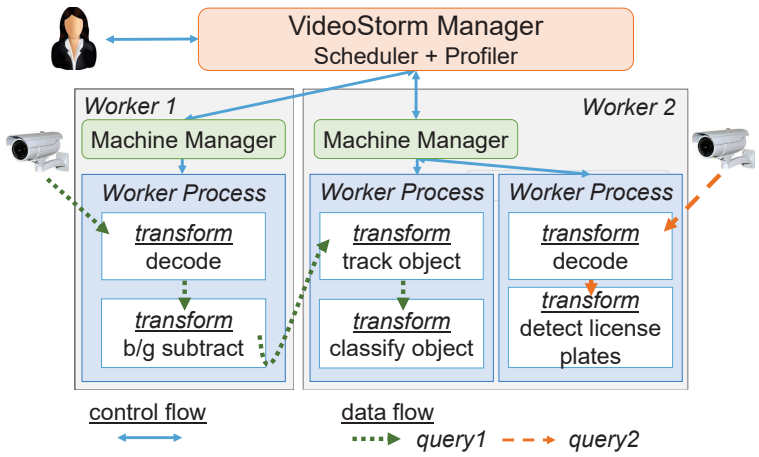


Figure 2.1: VideoStorm system architecture.

real video analytics queries over video datasets from live traffic cameras from several large cities. Our offline profiling consumes 3.5× less CPU resources compared to a basic greedy search. The online VideoStorm scheduler outperforms fair scheduling of resources [3, 65, 100] by as much as 80% in quality of queries and 7× in terms of lag.

2.1 System Description

In this section, we describe the high-level architecture of VideoStorm and the specifications for video queries.

2.1.1 VideoStorm Architecture

The VideoStorm cluster consists of a centralized *manager* and a set of *worker machines* that execute *queries*, see Figure 2.1. Every query is a DAG of *transforms* on *live video* that is continuously streamed to the cluster; each transform processes a time-ordered stream of messages (e.g., video frames) and passes its outputs downstream.

Figure 2.1 shows two example queries. One query runs across two machines; after decoding the video and subtracting the background, it sends the detected objects to another machine for tracking and classification. The other query for detecting license plates runs

```
"name": "LicensePlate",
"transforms": [
  {"id": "0",
   "class_name": "Decoder",
   "parameters": {
     "CameraIP": "134.53.8.8",
     "CameraPort": 8100,
     "@OutputResolution": "720P",
     "@SamplingRate": 0.75 }
  },
  {"id": "1",
   "input_transform_id": "0",
   "class_name": "OpenALPR",
   "parameters": {
     "@MinSize": 100,
     "@MaxSize": 1000,
     "@Step": 10 }
  } ]
```

Figure 2.2: VideoStorm query for license plate reader.

on a single machine. We assume there is sufficient bandwidth provisioned for cameras to stream their videos into the cluster.

Every worker machine runs a *machine manager* which start *worker processes* to host transforms. The machine manager periodically reports resource utilizations as well as status of the running transforms to the VideoStorm manager. The scheduler in the manager uses this information to allocate resources to queries. The VideoStorm manager and the machine managers are not on the query *data path*; videos are streamed directly to the decoding transforms and thereon between the transforms.

2.1.2 Video Queries Specification

Queries submitted to the VideoStorm manager are strung together as pipelines of *transforms*. Figure 2.2 shows a sample VideoStorm pipeline with two transforms. The first transform decodes the live video to produce frames that are pushed to the second transform to find license plate numbers using the OpenALPR library [29].

Each transform contains an `id` and `class_name` which is the class implementing the transform. The `input_transform_id` field specifies the transform whose output feeds into this transform, thus allowing us to describe a pipeline. VideoStorm allows arbitrary DAGs including multiple inputs and outputs for a transform. *Source* transforms, such as the "Decoder", do not specify input transform, but instead directly connect to the camera source (specified using IP and port number).

Each transform contains optional knobs (`parameters`); e.g., the minimum and maximum window sizes (in pixels) of license plates to look for and the step increments to search between these sizes for the OpenALPR transform (more in §2.4). Knobs whose values can updated dynamically start with the '@' symbol. The VideoStorm manager updates them as part of its scheduling decisions.

2.2 Making the Case for Resource Allocation

We make the case for resource management in video analytics clusters using a simple example (§2.2.1) and real-world video queries (§2.2.2).

2.2.1 Motivating Example

Cluster managers such as Yarn [3], Apollo [65], and Mesos [100] commonly divide resources among multiple queries based on resource fairness. Being agnostic to query quality and lag preferences, fair allocation is the best they can do. Instead, *scheduling for performance* leads to queries achieving better quality and lag.

The desirable properties of a scheduler for video analytics are: (1) allocate more resources to queries whose qualities will improve more, (2) allow queries with built-up lag during resource shortage to "catch up" in later processing, and (3) adjust query configuration based on the resource allocated.

(a) Query A

C	D	Q
A1	1	0.6
A2	2	0.7
A3	3	0.8

(b) Query B

C	D	Q
B1	1	0.1
B2	2	0.3
B3	3	0.9

(c) Fair allocation

		Query A					Query B				
Time	R	C	D	A	Q	L	C	D	A	Q	L
0	4	A2	2	2	0.7	-	B2	2	2	0.3	-
10	2	A1	1	1	0.6	-	B1	1	1	0.1	-
22	4	A2	2	2	0.7	-	B2	2	2	0.3	-

(d) Performance-based allocation

		Query A					Query B				
Time	R	C	D	A	Q	L	C	D	A	Q	L
0	4	A1	1	1	0.6	-	B3	3	3	0.9	-
10	2	A1	1	1	0.6	-	B3	3	1	0.9	-
22	4	A1	1	1	0.6	-	B2	2	3	0.3	8s
38	4	A1	1	1	0.6	-	B3	3	3	0.9	-

Table 2.1: Tables (a) and (b) show queries A and B with three configurations each, resource demand D and quality Q. Tables (c) and (d) show the time and capacity R, and for each query the chosen configuration C, demand D, allocation A, achieved quality Q, and lag L for the fair and performance-based schedulers. Notice in (d) that query B achieves higher quality between times 10 and 22 than with the fair scheduler in (c), and never lags beyond its permissible 8s.

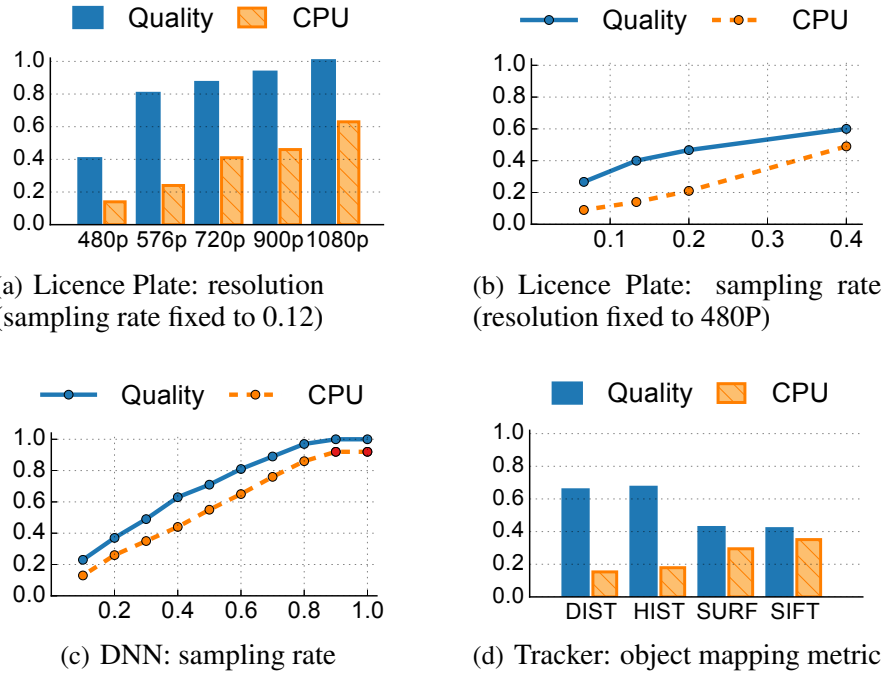
Tables 2.1(a) and 2.1(b) shows two example queries A and B with three knob configurations each (Ax and Bx, respectively). Query A's improvement in quality Q is less pronounced than B's for the same increase in resource demand D. Note that D is the resource to keep up with the incoming data rate. Query A cannot tolerate any lag, but B can tolerate up to 8 seconds of lag. Lag is defined as the difference between the time of the last-arrived frame and the time of the last-processed frame, i.e., how much time's worth of frames are queued-up unprocessed.

Let a single machine with resource capacity R of 4 run these two queries. Its capacity R drops to 2 after 10 seconds and then returns back to 4 after 12 more seconds (at 22 seconds). This drop could be caused by another high-priority job running on this machine.

Fair Scheduling. Table 2.1(c) shows the assigned configuration C, query demand D, resource allocation A, quality Q and lag L with a fair resource allocation. Each query selects the best configuration to keep up with the live stream (i.e., keeps its demand below its allocation). Using the fair scheduler, both queries get an allocation of 2 initially, picking configurations A2 and B2 respectively. Between times 10 to 22, when the capacity drops to 2, the queries get an allocation of 1 each, and pick configurations A1 and B1. At no point do they incur any lag.

Performance-based Scheduling. As Table 2.1(d) shows, a performance-based scheduler allocates resources of 1 and 3 to queries A and B at time 0; B can thus run at configuration B3, achieving higher quality compared to the fair allocation (while A's quality drops only by 0.1). This is because the scheduler realizes the value in providing more resources to B given its resource-quality profile.

At time 10 when capacity drops to 2, the scheduler allocates 1 unit of resource to each to the queries, but retains configuration B3 for B. Since resource demand of B3 is 3, but B has been allocated only 1, B starts to lag. Specifically, every second, the lag in processing will *increase* by $2/3$ of a second. However, query B will still produce results at quality 0.9, albeit delayed. At time 22, the capacity recovers and query B has built up a lag of 8 seconds. The scheduler allocates 3 resource units to B but switches it to configuration B2 (whose demand is only 2). This means that query B can now *catch up*—every second it can process 1.5 seconds of video. Finally, at time 38, all the lag has been eliminated and the scheduler switches B to configuration B3 (quality 0.9).



(a) Licence Plate: resolution (sampling rate fixed to 0.12)

(b) Licence Plate: sampling rate (resolution fixed to 480P)

(c) DNN: sampling rate

(d) Tracker: object mapping metric

Figure 2.3: Resource-quality profiles for real-world video queries including Licence Plate Reader, DNN Classifier, and Object Tracker. For simplicity, we plot one knob at a time.

The performance-based scheduler exhibited the three properties listed above. It allocated resources to optimize for quality and allowed queries to catch up to built-up lag, while accordingly adjusting their configurations.

2.2.2 Real-world Video Queries

Video analytics queries have many *knob configurations* that affect output quality and resource demand. We highlight the resource-quality profiles of four real-world queries—*license plate reader, car counter, DNN classifier, object tracker*—of interest to the cities we are partnering with and obtained videos from their *operational* traffic cameras (§2.7.1). For clarity, we plot one knob at a time and keep other knobs fixed. Quality is defined as the F1 score $\in [0, 1]$ (the harmonic mean between precision and recall [177]) with reference to a *labeled ground truth*.

License Plate Reader. The OpenALPR library [29] scans the video frame to detect potential plates and then recognizes the text on plates using optical character recognition. In general, using higher video resolution and processing each frame will detect the most license plates accurately. Reducing the resolution and processing only a subset of frames (e.g., sampling rate of 0.25) dramatically reduces resource demand, but can also reduce the quality of the output (i.e., miss or incorrectly read plates). Figures 2.3(a) and 2.3(b) plots the impact of resolution and sampling rate[2] on quality and CPU demand.

Car Counter. Resolution and sampling rate are knobs that apply to almost all video queries. A car counter monitors an "area of interest" and counts cars passing the area. In general, its results are of good quality even on videos with low resolution and sampling rates (plots omitted).

Deep Neural Network (DNN) Classifier. Vision processing is employing DNNs for key tasks including object detection and classification. Figure 2.3(c) profiles a Caffe [112] DNN model trained with the widely-used ImageNet dataset [82] to classify objects into $1,000$ categories. We see a uniform increase in the quality of the classification as well as resource consumption with the sampling rate. As DNN models get *compressed* [93, 94], reducing their resource demand at the cost of quality, the compression factor presents another knob.

Object Tracker. Finally, we have also profiled an object tracker. This query continuously models the "background" in the video, identifies foreground objects by subtracting the background, and tracks objects *across frames* using a mapping metric. The mapping metric is a key knob (Figure 2.3(d)). Objects across frames can be mapped to each other using metrics such as distance moved (DIST), color histogram similarity (HIST), or matched over SIFT [30] and SURF [31] features.

[2]Sampling rate of 0.75 drops every fourth frame from the video.

Resource-quality profiles based on knob configurations is intrinsic to video analytics queries. These queries typically identify "events" (like license plates or car accidents), and using datasets where these events are labeled, we can empirically measure precision and recall in identifying the events for different query configurations.

In contrast to approximate SQL query processing, there are no analytical models to estimate the relationship between resource demand and quality of video queries and it *depends on the specific video feeds*. For example, reducing video resolution may not reduce OpenALPR quality if the camera is zoomed in enough. Hence queries need to be *profiled* using representative video samples.

2.2.3 Summary and Challenges

Designing a scheduler with the desirable properties in §2.2.1 for real-world video queries (§2.2.2) is challenging.

First, the configuration space of a query can be large and there are no analytical models to estimate the resource demand and result quality of each configuration.

Second, trading off between the lag and quality goals of queries is tricky, making it challenging to define scheduling objectives *across all queries in the cluster*.

Third, resource allocation across all queries in the cluster each with many configurations is computationally intractable, presenting scalability challenges.

2.3 VideoStorm Design Overview

The VideoStorm scheduler is split into *offline profiling* and *online scheduling* phases (Figure 2.4). In the offline phase, for every query, we efficiently generate its *resource-quality profile* – a small number of configurations on the Pareto curve of the profile, §2.4. This dramatically reduces the configurations to be considered by the scheduler.

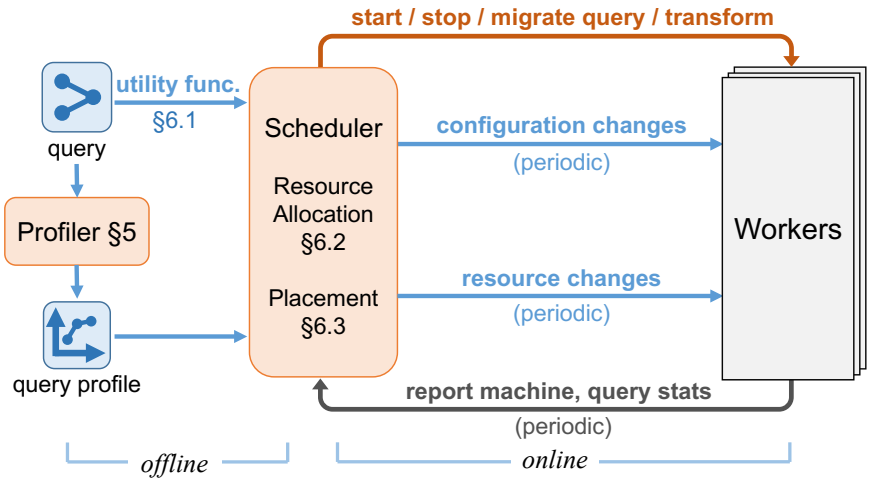


Figure 2.4: VideoStorm scheduler components.

In the online phase, the scheduler periodically (e.g., every second) considers all running queries and adjusts their resource allocation, machine placement, and configurations based on their profiles, changes in demand and/or capacity (see Figure 2.4). We encode the quality and lag requirements of *each individual query* into its utility function, §2.5.1. The performance goal across *all queries in a cluster* is specified either as maximizing the minimum utility or the sum of utilities, §2.5.2 and §2.5.3.

2.4 Resource-Quality Profile Estimation

When a user submits a new query, we start running it immediately with a default profile (say, from its previous runs on other cameras), while at the same time we run the query through the offline profiling phase. The query profiler has two goals. 1) Select a small subset of configurations (Pareto boundary) from the resource-quality space, and 2) Compute the query profile, $\mathscr{P}_k$, i.e., the resource demand and result quality of the selected configurations. The profile is computed either against a labeled dataset or using the initial parts of the video relative to a "golden" query configuration which might be expensive but is known to produce high-quality results.

2.4.1 Profile estimation is expensive

We revisit the license plate reader query from §2.2.2 in detail. As explained earlier, frame *resolution* and *sampling rate* are two important knobs. The query, built using the OpenALPR library [29], scans the image for license plates of size *MinSize*, then multiplicatively increases the size by *Step*, and keeps repeating this process until the size reaches *MaxSize*. The set of potential license plates is then sent to an optical character recognizer.

We estimate the quality of each knob configuration (i.e., combination of the five knobs above) on a labeled dataset using the F1 score [177], the harmonic mean between precision and recall, commonly used in machine learning; 0 and 1 represent the lowest and highest qualities. For example, increasing *MinSize* or decreasing *MaxSize* reduces the resources needed but can miss some plates and decrease quality.

Figure 2.5 shows a scatter plot of resource usage vs. quality of 414 configurations generated using the five knobs. There is four orders of magnitude of difference in resource usage; the most expensive configuration used all frames of a full HD resolution video and would take over 2.5 hours to analyze a 1 minute video on 1 core. Notice the vast spread in quality among configurations with similar resource usage as well as the spread in resource usage among configurations that achieve similar quality.

2.4.2 Greedy exploration of configurations

We implement a greedy local search to identify configuration with high quality (Q) and low demand (D); see Table 2.2. Our *baseline profiler* implements hill-climbing [162]; it selects a random configuration c, computes its quality $Q(c)$ and resource demand $D(c)$ by running the query with c on a *small subset* of the video dataset, and calculates $X(c) = Q(c) - \beta D(c)$ where β trades off between quality and demand. Next, we pick a neighbor configuration n (by changing the value of a *random* knob in c). If $X(n) > X(c)$, then n is better than c in quality or resource demand (or both); we set $c = n$ and repeat. When we cannot find a

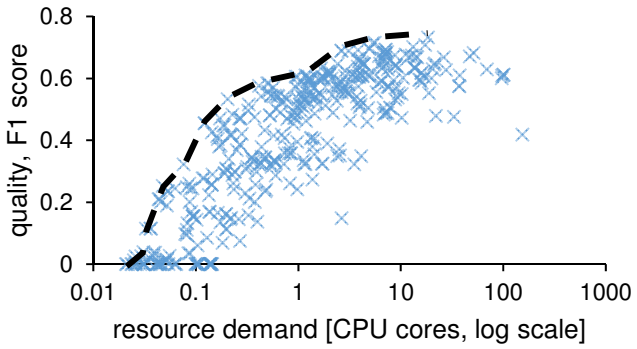


Figure 2.5: Resource-quality for license plate query on a 10 minute video (414 configurations); x-axis is resource demand to keep up with live video. Generating this took 20 CPU *days*. The black dashed line is the Pareto boundary.

better neighbor (i.e., our exploration indicates that we are near a local optimum), we repeat by picking another random c.

Several enhancements significantly increase the efficiency of our search. To avoid starting with an expensive configuration and exploring its neighbors, (which are also likely to be expensive, thus wasting CPU), we pick k random configurations and start from the one with the highest $X(c)$. We found that using even $k = 3$ can successfully avoid starting in an expensive part of the search space. Second, we cache *intermediate results* in the query's DAG and reuse them in evaluating configurations with overlapping knob values.

While our simple profiler is sufficiently efficient for our purpose, sophisticated hyperparameter searches (e.g., [166]) can potentially further improve its efficiency.

Pareto boundary. We are only interested in a small subset of configurations that are on the *Pareto boundary* $\mathscr{P}$ of the resource-quality space. Let $Q(c)$ be the quality and $D(c)$ the resource demand under configuration c. If c_1 and c_2 are two configurations such that $Q(c_1) \geq Q(c_2)$ and $D(c_1) \leq D(c_2)$, then c_2 is not useful in practice; c_1 is better than c_2 in both quality and resource demand. The dashed line in Figure 2.5 shows the Pareto boundary of such configurations for the license plate query. We extract the Pareto boundary of the explored configurations and call it the resource-quality profile $\mathscr{P}$ of the query.

Term	Description
$\mathscr{C}_k$	set of configurations of query k
$\mathscr{P}_k$	profile of query k
$c_k \in \mathscr{C}_k$	specific configuration of query k
$Q_k(c)$	quality under configuration c
$D_k(c)$	resource demand under configuration c
$L_{k,t}$	measured lag at time t
U_k	utility
Q_k^M	(min) quality goal
L_k^M	(max) lag goal
a_k	resources allocated

Table 2.2: Notations used, for query k.

We can generate the same profile as the baseline profiler on the license plate query with 3.5× less CPU resources (i.e., 5.4 CPU hours instead of 19 CPU hours).

2.5 Resource Management

In the *online phase*, the VideoStorm cluster scheduler considers the utilities of individual queries and the cluster-wide performance objectives (defined in §2.5.1) and periodically performs two steps: resource allocation and query placement. In the resource *allocation step*, §2.5.2, the scheduler assumes the cluster is an aggregate bin of resources and uses an efficient heuristic to maximize the cluster-wide performance by adjusting query allocation and configuration. In the query *placement step*, §2.5.3, the scheduler places new queries to machines in the cluster and considers migrating existing queries.

2.5.1 Utility: Combining Quality and Lag

Each query has preferences on the desired quality and lag. What is the *minimum* quality goal (Q^M)? How much does the query benefit from higher quality than the goal? What is the *maximum* lag (L^M) it can tolerate and how sensitive are violations to this goal? (See

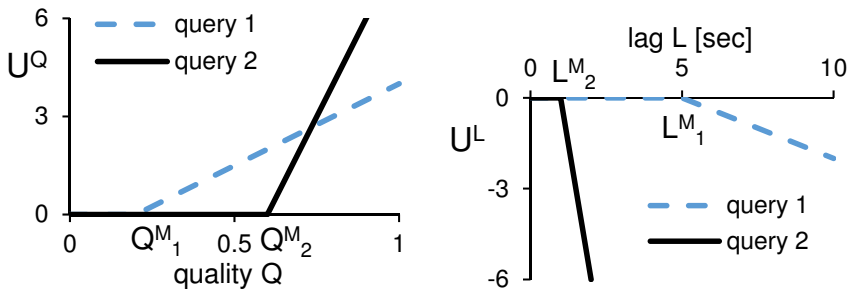


Figure 2.6: Examples for the second (U^Q) and third terms (U^L) in Equation 2.1.
(Left) Query 1's quality goal is relatively lenient, $Q_1^M = 0.2$, but its utility grows slowly with increase in quality beyond Q_1^M. Query 2 is more stringent, $Q_2^M = 0.6$, but its utility grows sharply thereon.
(Right) Query 1 has lag target of $L_1^M = 5$ beyond which it incurs a penalty. Query 2 has a stricter lag goal of $L_2^M = 1$ and also its utility drops much faster with increased lag.

Table 2.2 for notations.) We encode these preferences in utility functions, an abstraction used extensively in economics [139, 161] and computer systems [46, 113].

Our utility function for a query has the following form, where $(x)_+$ is the positive part of x. We omit the query index k for clarity.

$$\begin{aligned} U(Q,L) &= U^B + U^Q(Q) + U^L(L) \\ &= U^B + \alpha^Q \cdot (Q - Q^M)_+ - \alpha^L \cdot (L - L^M)_+ \end{aligned} \tag{2.1}$$

U^B is the "baseline" utility for meeting the quality and lag goals (when $Q = Q^M$ and $L = L^M$). The second term U^Q describes how the utility responds to achieved quality Q above Q^M, the soft quality goal; the multiplier α^Q and Q^M are query-specific and set based on the application analyzing the video. Results with quality below Q^M are typically not useful to the users.

The third term, U^L, represents the penalty for results arriving later than the maximum lag goal of L^M. [3] Recall that lag is the difference between the current time and the arrival time of the last processed frame, e.g., if at time 10:30 we process a frame that arrived at

[3] Multiplier α^L is in (1/second), making U^L dimensionless like U^Q.

10:15, the lag is 15 minutes. Similar to latency SLOs in clusters, there is no bonus for lag being below L^M. See Figure 2.6 for examples of U^Q and U^L in queries.

Scheduling objectives. Given utilities of individual queries, how do we define utility or *performance* of the whole cluster? Previous work has typically aimed to maximize the minimum utility [126, 136] or sum of utilities [126, 133], which we adopt. When deployed as a "service" in the public cloud, utility will represent the revenue the cluster operator generates by executing the query; penalties and bonuses in utility translate to loss and increase in revenue. Therefore, *maximizing the sum of utilities* maximizes revenue. In a private cluster that is shared by many cooperating entities, achieving fairness is more desirable. Maximally improving the utility of the worst query provides *max-min fairness over utilities*.

To simplify the selection of utility functions in practical settings, we can provide only a few options to choose from. For example, the users could separately pick the minimum quality (40%, 60%, or 80%) and the maximum lag (1, 10, or 60 minutes) for a total of nine utility function *templates*. Users of cloud services already make similar decisions; for example, in Azure Storage [67], they separately select data redundancy (local, zone, or geo-distributed) and data access pattern (hot vs. cool).

2.5.2 Resource Allocation

Given a profile $\mathscr{P}_k$ and a utility function U_k for each query k, the scheduler allocates resources a_k to the queries and picks their query configuration ($c_k \in \mathscr{P}_k$). The scheduler runs periodically (e.g., every few seconds) and reacts to arrival of new queries, changes in query demand and lag, and changes in resource capacity (e.g., due to other high-priority non-VideoStorm jobs).

Scheduling Using Model-Predictive Control

The scheduler aims to maximize the minimum or sum of query utilities, which in turn depend on their quality and lag. A key point to understand is that while we can *near-instantaneously* control query quality by adjusting its configuration, query lag *accumulates* over time if we allocate less resources than query demand.

Because of this accumulation property, the scheduler cannot optimize the *current performance*, but only aims to improve performance in the *near future*. We formulate the scheduling problem using the Model-Predictive Control (MPC [142]) framework; where we model the cluster performance over a short time horizon T as a function of query configuration and allocation. In each step, we select the configuration and allocation to maximize performance over the near future (described in detail in §2.5.2).

To predict future performance, we need to predict query lag; we use the following formula:

$$L_{k,t+T}(a_k, c_k) = L_{k,t} + T - T\frac{a_k}{D_k(c_k)} \tag{2.2}$$

We plug in the predicted lag $L_{k,t+T}$ into the utility function (Equation 2.1) to obtain the predicted utility.

Scheduling Heuristics

We describe resource allocation assuming each query to contain only one transform, which we relax in §2.5.4.

Maximizing sum of utilities. The optimization problem for maximizing sum of utilities over time horizon T is as follows. Sum of allocated resources a_k cannot exceed cluster

resource capacity R.

$$\begin{aligned} \max_{a_k, c_k \in \mathscr{P}_k} \quad & \textstyle\sum_k U_k(Q_k(c_k), L_{k,t+T}) \\ s.t. \quad & \textstyle\sum_k a_k \leq R \end{aligned} \tag{2.3}$$

Maximizing the sum of utilities is a variant of the knapsack problem where we are trying to include the queries at different allocation and configuration to maximize the total utility. The maximization results in the best distribution of resources (as was illustrated in §2.2.1).

When including query k at allocation a_k and configuration c_k, we are *paying* cost of a_k and *receiving value* of $u_k = U_k(Q_k(c_k), L_{k,t+T})$. We employ a greedy approximation based on [80] where we prefer queries with highest value of u_k/a_k; i.e., we receive the largest increase in utility normalized by resource spent.

Our heuristic starts with $a_k = 0$ and in each step we consider increasing a_i (for all queries i) by a small Δ (say, 1% of a core) and consider all configurations of $c_i \in \mathscr{P}_i$. Among these options, we select query i (and corresponding c_i) with largest increase in utility.[4] We repeat this step until we run out of resources or we have selected the best configuration for each query. (Since we start with $a_k = 0$ and stop when we run out of resources, we will not end up with infeasible solutions.)

Maximizing minimum utility. Below is the optimization problem to maximize the minimum utility predicted over a short time horizon T. We require that all utilities be $\geq u$ and we maximize u.

$$\begin{aligned} \max_{a_k, c_k \in \mathscr{P}_k} \quad & u \\ s.t. \quad & \forall k : U_k(Q_k(c_k), L_{k,t+T}) \geq u \\ & \textstyle\sum_k a_k \leq R \end{aligned} \tag{2.4}$$

[4] We use a *concave* version of the utility functions obtained using linear interpolation to ensure that each query has a positive increase in utility, even for small Δ.

We can improve u only by improving the utility of the worst query. Our heuristic is thus as follows. We start with $a_k = 0$ for all queries. In each step, we select query $i = \arg\min_k U_k(Q_k(c_k), L_{k,t+T})$ with the lowest utility and increase its allocation by a small Δ, say 1% of a core. With this allocation, we compute its best configuration c_i as $\arg\max_{c \in \mathscr{P}_i} U_i(Q_i(c), L_{i,t+T})$. We repeat this process until we run out of resources or we have picked the best configuration for each query.

2.5.3 Query Placement

After determining resource allocation and configuration of each query, we next describe the placement of new queries and migration of existing queries. We quantify the suitability of placing a query q on machine m by computing a score for each of the following goals: high utilization, load balancing, and spreading low-lag queries.

(i) Utilization. High utilization in the cluster can be achieved by *packing* queries in to machines, thereby minimizing fragmentation and wastage of resources. Packing has several well-studied heuristics [91, 152]. We define *alignment* of a query relative to a machine using a weighted dot product, p, between the vector of machine's available resources and the query's demands; $p \in [0,1]$.

(ii) Load Balancing. Spreading load across the cluster ensures that each machine has spare capacity to handle changes in demand. We therefore prefer to place q on a machine m with the smallest utilization. We capture this in score $b = 1 - \frac{M+D}{M_{max}} \in [0,1]$, where M is the current utilization of machine m and D is demand of query q.

(iii) Lag Spreading. Not concentrating many low-lag queries on a machine provides *slack* to accumulate lag for some queries when resources are scarce, *without* having to resort to migration of queries or violation of their lag goal L^M. We achieve this by maintaining *high average* L^M on each machine. We thus compute score $l \in [0,1]$ as the average L^M *after* placing q on m.

The final score $s_{q,m}$ is the average of the three scores. For each new query q, we place it on a machine with the largest $s_{q,m}$. For each existing query q, we migrate from machine m_0 to a new machine m_1 only if its score improves substantially; i.e., $s(q,m_1) - s(q,m_0) > \tau$.

2.5.4 Enhancements

Incorrect resource profile. The profiled resource demand of a query, $D_k(c_k)$, might not exactly correspond to the actual query demand, e.g., when demand depends on video content. Using incorrect demand can negatively impact scheduling; for example, if $D_k(c) = 10$, but actual usage is $R_k = 100$, the scheduler would estimate that allocating $a_k = 20$ would reduce query lag at the rate of $2\times$, while the lag would actually *grow* at a rate of $5\times$. To address this, we keep track of a running average of *mis-estimation* $\mu = R_k/D_k(c)$, which represents the multiplicative error between the predicted demand and actual usage. We then incorporate μ in the lag predictor from Equation 2.2, $L_{k,t+T}(a_k, c_k) = L_{k,t} + T - T\frac{a_k}{D_k(c_k)}(\frac{1}{\mu})$.

Machine-level scheduling. As most queries fit on a single machine, we can respond to changes in demand or lag at the machine-level, without waiting for the cluster-wide decisions. We therefore execute the allocation step from §2.5.2 on each machine, which makes the scheduling logic much more scalable. The cluster-wide scheduler still runs the allocation step, but only for the purposes of determining query placement and migration.

DAG of transforms. Queries consisting of a DAG of transforms could be placed across multiple machines. We first distribute the query resource allocation, a_k, to *individual transforms* based on per-transform resource demands. We then place individual transforms to machines as described in §2.5.3 while accounting for the expected data flow across machines and network link capacities.

2.6 VideoStorm Implementation

We now discuss VideoStorm's key implementation details and the interfaces implemented by transforms.

2.6.1 Implementation Details

In contrast to widely-deployed cluster frameworks like Yarn [3], Mesos [100] and Cosmos [65], we highlight the differences in VideoStorm's design. First, VideoStorm takes the list of knobs, resource-quality profiles and lag goals as inputs to allocate resources. Second, machine-level managers in the cluster frameworks *pull* work, whereas the VideoStorm manager *pushes* new queries and configuration changes to the machine-managers. Finally, VideoStorm allows machine managers to autonomously handle short-term fluctuations (§2.5.4)

Flow control. We implemented flow control across transforms of a query to minimize the buffering inside the query pipeline, and instead push queuing of unprocessed video to the *front* of the query. This helps for two reasons. First, decoded frames can be as much as 300× larger than the encoded video (from our benchmarks on HD videos). Buffering these frames will significantly inflate memory usage while spilling them to disk affects overall performance. Second, buffering at the front of query enables the query to respond promptly to configuration changes. It prevents frames from being processed by transforms with old inconsistent knob values.

Migration. As described in §2.5.3, VideoStorm migrates queries depending on the load in the cluster. We implement a simple "start-and-stop" migration where we start a copy of a running query/transform on the target machine, duplicate its input stream to the copy, and stop the old query/transform after a short period. The whole process of migration is

data-lossless and takes roughly a second (§2.7.3), so the overhead of duplicated processing during the migration is very small.

Resource Enforcement. VideoStorm uses Job Objects [39] for enforcing allocations. Similar to Linux Containers [22], Job Objects allow controlling *and resizing* the CPU/memory limits of running processes.

2.6.2 Interfaces for Query Transforms

Transforms implement simple interfaces to process data and exchange control information.

- *Processing*. Transforms implement `byte[] Process(header, data)` method. `header` contains metadata such as frame id and timestamp. `data` is the input byte array, such as decoded frame. The transform returns another byte array with its result, such as the detected license plate. Each transform maintains its own state, such as the background model.
- *Configuration*. Transforms can also implement `Update(key, value)` to set and update knob values to change query configuration at runtime.

2.7 Evaluation

We evaluate the VideoStorm prototype (§2.6) using a cluster of 101 machines on Microsoft Azure with real video queries and video datasets. Our highlights:

1. VideoStorm outperforms the fair scheduler by 80% in quality of outputs with 7× better lag. (§2.7.2)
2. VideoStorm is robust to errors in query profiles and allocates nearly the same as correct profiles. (§2.7.3)
3. VideoStorm scales to thousands of queries with little systemic execution overheads. (§2.7.4)

2.7.1 Setup

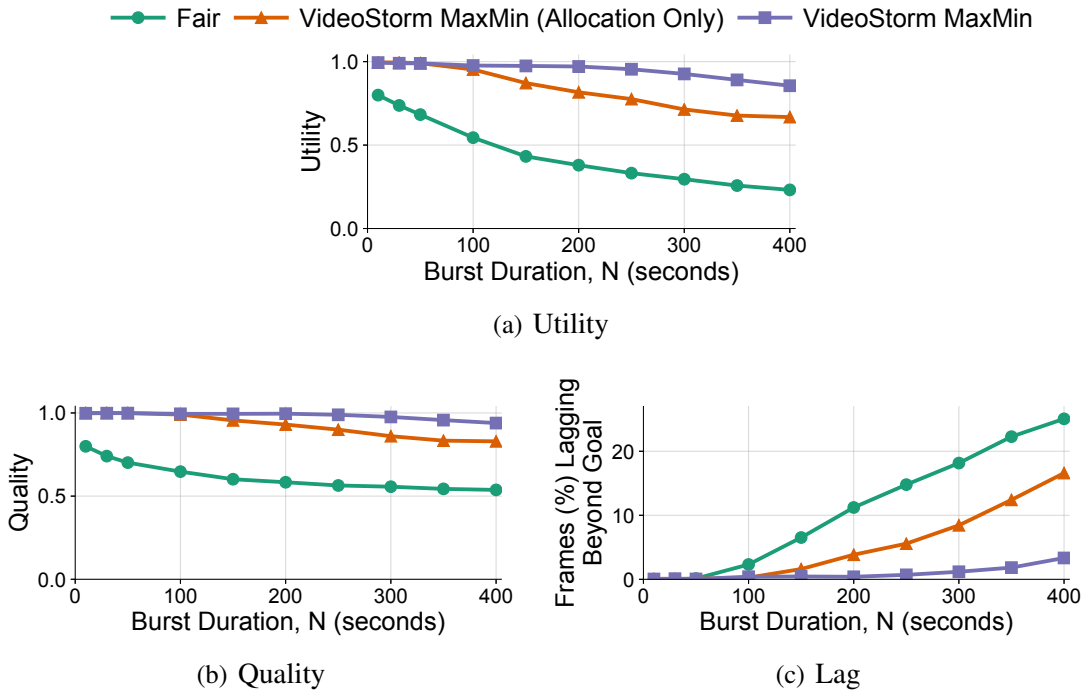
Video Analytics Queries. We evaluate VideoStorm using four types of queries described and profiled in §2.2.2—license plate reader, car counter, DNN classifier, object tracker. These queries are of major interest to the cities we are partnering with in deploying our system.

Video Datasets. The above queries run on video datasets obtained from real and operational traffic cameras in Bellevue and Seattle cities for two months (Sept.–Oct., 2015). In our experiments, we stream the recorded videos at their original frame-rate (14 to 30 fps) and resolution (240P to 1080P) thereby mimicking live video streams. The videos span a variety of conditions (sunny/rainy, heavy/light traffic) that lead to variation in their processing workload. We present results on multiple different snippets from the videos.

Azure Deployment. We deploy VideoStorm on 101 `Standard_D3_v2` instances on Azure's West-US cluster [13]. `Standard_D3_v2` instances contain 4 cores of the 2.4GHz Intel Xeon processor and 14GB RAM. One machine ran the VideoStorm global manager on which no queries were scheduled.

Baseline. We use the work-conserving fair scheduler as our baseline. It's the widely-used scheduling policy for cluster computing frameworks like Mesos [100], Yarn [3] and Cosmos [65]. When a query, even at its best configuration, cannot use its fair share, it distributes the excess resources among the other queries. The fair scheduler places the same number of queries evenly on all available machines in a round-robin fashion.

Metric. The three metrics of interest to us are quality, frames (%) exceeding the lag goal in processing, and utility (§2.5.1). We compare the improvement (%); if a metric (say, quality) with VideoStorm and the fair scheduler is X_V and X_f, we measure $\frac{X_V - X_f}{X_f} \times 100\%$.



(a) Utility

(b) Quality

(c) Lag

Figure 2.7: VideoStorm outperforms the fair scheduler as the *duration* of burst of queries in the experiment is varied. Without its placement but only its allocation ("VideoStorm MaxMin (Allocation Only)"), its performance drops by a third.

2.7.2 Performance Improvements

Our workload consists of a mix of queries with lenient and stringent goals. We start with a set of 300 queries picked from the four types (§2.7.1) on 300 distinct video datasets at the beginning of the experiment. 60% of these queries have a lag goal L^M of 20s while the remaining are more lenient with a lag goal of 300s. All of them have a quality goal Q^M of 0.25. We set the lag multiplier $\alpha^L = 1$ for these long-lived video analyses.

Burst of N seconds: At a certain point, a burst of 200 license plate queries arrive and last for N seconds (which we will vary). These queries have a lag goal Q^L of 20s, a high quality goal (1.0), and higher $\alpha^L = 2$. They mimic short-term deployment of queries like AMBER Alerts with stringent accuracy and lag goals. We evaluate VideoStorm's reaction to the burst of queries up to several minutes; note that the improvements will carry over when tolerant delay and bursts are much longer.

Maximize the Minimum Utility (MaxMin)

We ran a series of experiments with burst duration N from 10 to 400 seconds. Figure 2.7(a) plots the minimum query utility achieved in each of the experiments, when VideoStorm maximizes the minimum utility (§2.5.2). For each point in the figure, we obtain the minimum utility, quality and lag over an interval that includes a minute before and after the N second burst. VideoStorm's utility ("VideoStorm-MaxMin") drops only moderately with increasing burst duration. Its placement and resource allocations ensure it copes well with the onset of and during the burst. Contrast with the fair scheduler's sharp drop with N.

The improvement in utility comes due to smartly accounting for the resource-quality profile and lag goal of the queries; see Figures 2.7(b) and 2.7(c). Quality (F1 score [177]; $\in$ [0, 1]) with the fair scheduler is 0.2 lower than VideoStorm to begin with, but reduces significantly to nearly 0.5 for longer bursts (higher N), while quality with VideoStorm stays at 0.9, or nearly 80% better. The rest of VideoStorm's improvement comes by ensuring that despite the accumulation in lag, fewer than 5% of the frames exceed the query's lag goal whereas with the fair scheduler it grows to be 7$\times$ worse.

How valuable is VideoStorm's placement? Figure 2.7 also shows the "VideoStorm MaxMin (Allocation Only)" graphs which lie in between the graphs for the fair scheduler and VideoStorm. As described in §2.5.3, VideoStorm first decides the resource allocation and then places them onto machines to achieve high utilization, load balancing and spreading of lag-sensitive and lag-tolerant queries. As the results show, not using VideoStorm's placement heuristic (instead using our baseline's round-robin placement) considerably lowers VideoStorm's gains.

Figure 2.8(top) explains VideoStorm's gains by plotting the allocation of CPU cores in the cluster over time, for burst duration $N = 150$s. We group the queries into three categories — the burst of queries with 20s lag goal and quality goal of 1.0, those with 20s lag goal, and 300s lag goal (both with quality goal of 0.25). We see that VideoStorm

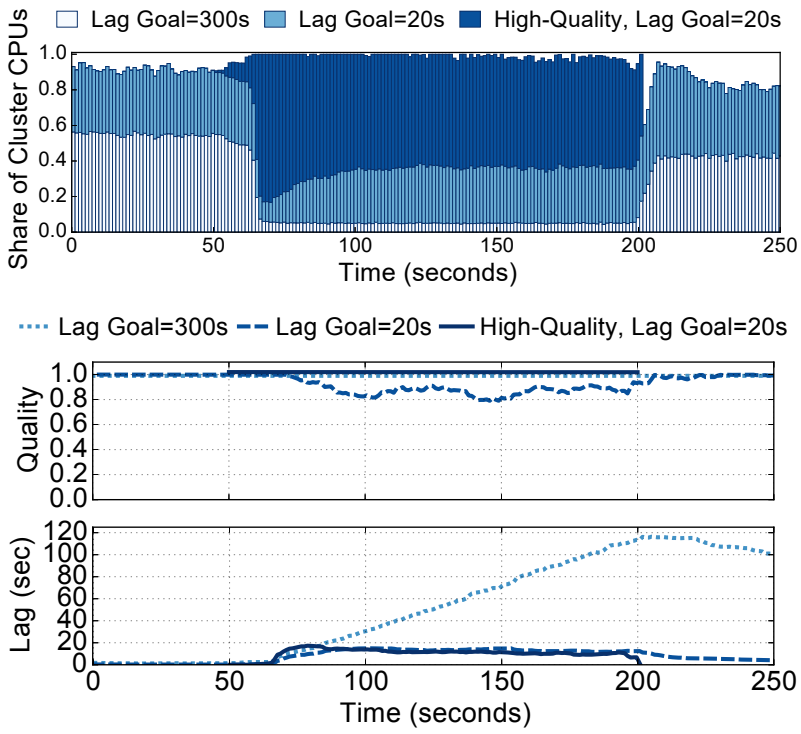


Figure 2.8: (Top) CPU Allocation for burst duration $N = 150$s, and (bottom) quality and lag averaged across all queries in each of the three categories.

adapts to the burst and allocates nearly 60% of the CPU cores in the cluster to the burst of license plate queries which have a high quality and tight lag goals. VideoStorm also delays processing of lag-tolerant queries (allocating less than 10% of CPUs). Figure 2.8(bottom) shows the resulting quality and lag, for queries in each category. We see that because the delay-tolerant queries have small allocation, their lag grows but stays below the goal. The queries with 20s lag goal reduce their quality to adapt to lower allocation and keep their lag (on average) within the bound.

Impact of α^L. Figure 2.9 plots the distinction in treatment of queries with the same lag goal (L^M) but different α^L and quality goals. While the figure on the left shows that VideoStorm does not drop the quality of the query with $Q^M = 1.0$, it also respects the difference in α^L; fewer frames of the query with $\alpha^L = 2$ lag beyond the goal of 20s (right). This is an example of how utility functions encode priorities.

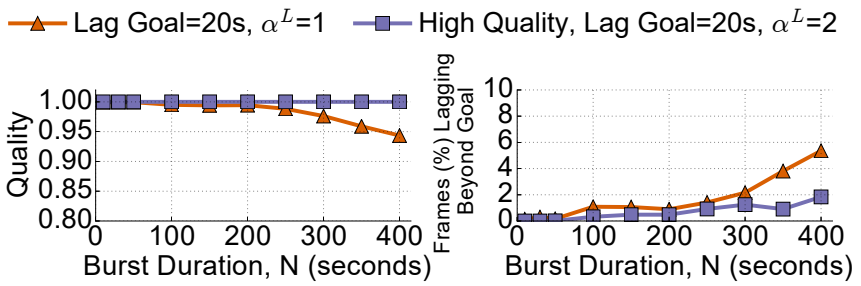
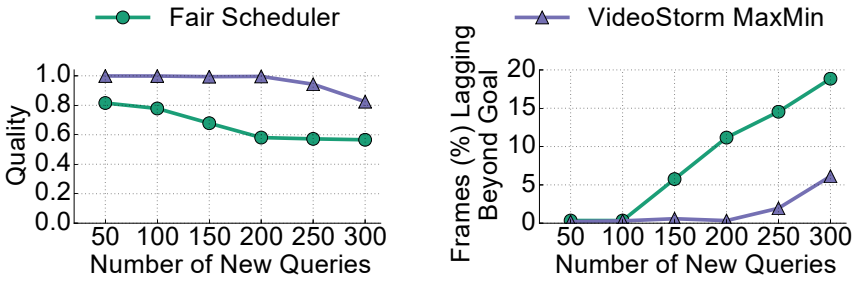


Figure 2.9: Impact of α^L. Queries with higher α^L have fewer overdue frames.



(a) Quality (b) Lag

Figure 2.10: VideoStorm vs. fair scheduler as the number of queries in the burst during the experiment is varied.

Maximize the Total Utility (MaxSum)

Recall from §2.5.2 that VideoStorm can also maximize the *sum* of utilities. We measure the *average* utility, quality, and frames (%) exceeding the lag goal; maximizing for the total utility and average utility are equivalent. VideoStorm achieves 25% better quality and 5× better lag compared with the fair scheduler.

Per Query Performance. While MaxMin scheduling, as expected, results in all the queries achieving similar quality and lag, MaxSum priorities between queries as the burst duration increases. Our results show that the license plate query, whose utility over its resource demand is relatively lower, is de-prioritized with MaxSum (reduced quality as well as more frames lagging). With its high quality (1.0) and low lag (20s) goals, the scheduler has little leeway. The DNN classifier, despite having comparable resource demand does not suffer from a reduction in quality because of its tolerance to lag (300s).

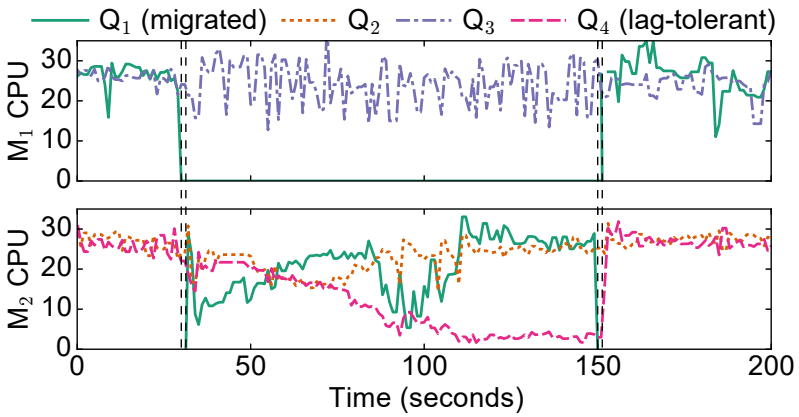


Figure 2.11: Q_1 migrated between M_1 and M_2. Resource for the only lag-tolerant query Q_4 (on M_2) is reduced for Q_1.

Varying the Burst Size

We next vary the *size* of the burst, i.e., number of queries that arrive in the burst. Note that the experiments above had varied the *duration* of the burst but with a fixed size of 200 queries. Varying the number of queries in the burst introduces different dynamics and reactions in VideoStorm's scheduler. We fix the burst duration to 200s. Figure 2.10 plots the results. The fair allocation causes much higher fraction of frames to exceed the lag goal when the burst size grows. VideoStorm better handles the burst and consistently performs better. Note that beyond a burst of 200 queries, resources are insufficient even to satisfy the lowest configuration (least resource demand), causing the degradation in Figure 2.10(b).

2.7.3 VideoStorm's Key Features

We now highlight VideoStorm's migration of queries and accounting for errors in the resource demands.

Migration of Queries

Recall from §2.5.3 and §2.6 that VideoStorm migrates queries when necessary. We evaluate the value of migration by making the following addition to our experiment described at the

beginning of §2.7.2. During the experiment, we allocate half the resources in 50% of our machines to other *non*-VideoStorm jobs. After a few minutes, the non-VideoStorm jobs complete and leave. Such jobs will be common when VideoStorm is co-situated with other frameworks in clusters managed by Yarn [3] or Mesos [100]. We measure the migration time, and compare the performance with and without migration.

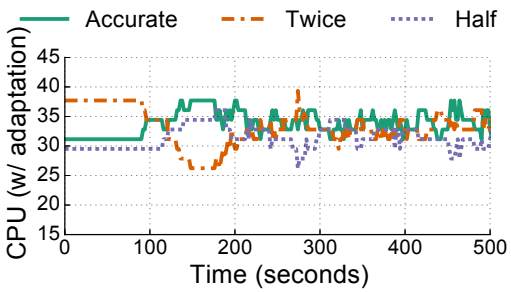
Figure 2.11 plots the timeline of two machines, M_1 and M_2; M_1 where a non-VideoStorm job was scheduled and M_2 being the machine *to* which a VideoStorm query Q_1, originally on M_1, was migrated. Q_1 shifts from running on M_1 to M_2 in only 1.3s. We migrate Q_1 back to M_1 when the non-VideoStorm job leaves at $\sim 150s$.

Shifting Q_1 to M_2 (and other queries whose machines were also allocated non-VideoStorm jobs, correspondingly) ensured that we did not have to degrade the quality or exceed the lag goals. Since our placement heuristic carefully spread out the queries with lenient and stringent lag goals (§2.5.3), we ensured that each of the machines had sufficient *slack*. As a result, when Q_1 was migrated to M_2 which already was running Q_2 and Q_4, we could delay the processing of the lag-tolerant Q_4 *without* violating any lag goals. The allocations of these delayed queries were ramped up for them to process their backlog as soon as the queries were migrated back.

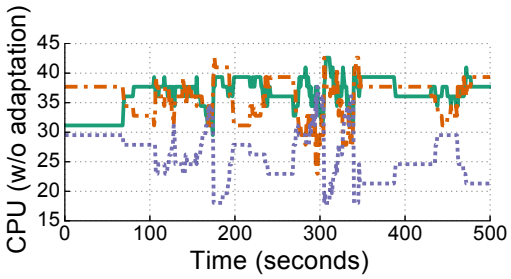
As a consequence, the quality of queries with migration is 12% better than without migration. Crucially, 18× more frames (4.55% instead of 0.25%) would have exceeded the lag goal without migration.

Handling Errors in Query Profile

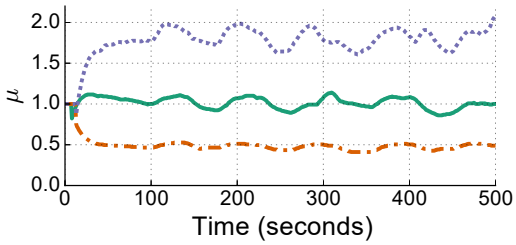
VideoStorm deals with difference between the resource demands in the resource-quality profile and the actual demand by continuously monitoring the resource consumption and adapting to errors in profiled demand (μ in §2.5.4). We now test the effectiveness of our correction.



(a) With Adaptation



(b) Without Adaptation



(c) μ Over Time

Figure 2.12: We show three queries on a machine whose resource demands in their profiles are synthetically doubled, halved, and unchanged. By learning the proportionality factor μ (2.12(c)), our allocation adapts and converges to the right allocations (2.12(a)) as opposed to without adaptation (2.12(b)).

We synthetically introduce *errors* in our profiles, as if they were profiles with errors, and use the erroneous profiles for our resource allocation. Consequently, the actual resource demands when the query executes do not match. In the workload above, we randomly make the profile to be half the actual resource demand for a third of the queries, twice the demand for another third, and unchanged (accurate) for the rest. VideoStorm's adaptive correction ensures that the quality and lag of queries with erroneous profiles are nearly 99.6% of results obtained *if* the profiles were perfectly accurate.

Action	Mean Duration (ms)	Standard Deviation (ms)
Start Transform	60.37	3.96
Stop Transform	3.08	0.47
Config. Change	15	2.0
Resource Change	5.7	1.5

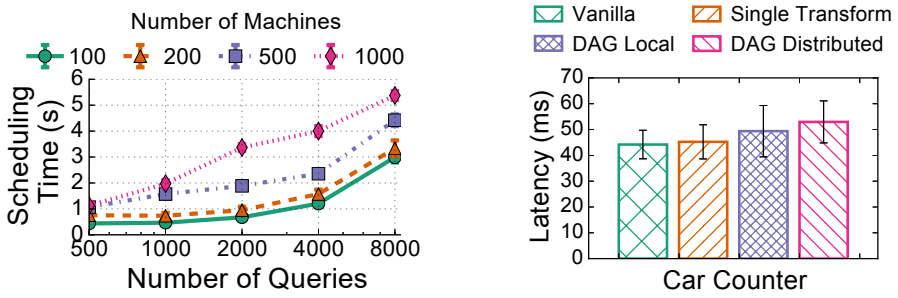
Table 2.3: Latency of VideoStorm's actions.

In Figure 2.12, we look at a single machine where VideoStorm placed three license plate queries, one each of the three different error categories. An ideal allocation (in the absence of errors) should be a third of the CPU to each of the queries. Figure 2.12(b), however, shows how the allocation is far from converging towards the ideal *without* adaptation, because erroneous profiles undermine the precision of utility prediction. In contrast, with the adaptation, despite the errors, resource allocations converge to and stay at the ideal (Figure 2.12(a)). This is because the μ values for the queries with erroneous profiles are correctly learned as 2 and 0.5; the query without any error introduced its profile has its μ around 1 (Figure 2.12(c)).

2.7.4 Scalability and Efficiency

Latency of VideoStorm's actions. Table 2.3 shows the time taken for VideoStorm to start a new transform (shipping binaries, process startup), stop a transform, and change a 100-knob configuration and resource allocation of 10 running queries. We see that VideoStorm allows for near-instantaneous operations.

Scheduling Decisions. Figure 2.13(a) plots the time taken by VideoStorm's scheduler. Even with thousands of queries, VideoStorm make its decisions in just a few seconds. This is comparable to the scalability of schedulers in big data clusters, and video analytics clusters are unlikely to exceed them in the number of queries. Combined with the low latency of actions (Table 2.3), we believe VideoStorm is sufficiently scalable and agile.



(a) Scheduling Scalability (b) Overheads

Figure 2.13: Overheads in scheduling and running queries.

Transform Overheads. Finally, we measure the overhead of running a vision algorithm inside VideoStorm. We compare the latency in processing a frame while running as a vanilla process, inside a single transform, as a DAG of transforms on one machine, and as a DAG distributed across machines. Figure 2.13(b) shows that the overheads are limited. Running as a single transform, the overhead is $< 3\%$. When possible, VideoStorm places the transforms of a query DAG locally on one machine.

2.8 Related Work on Stream Processing Systems

Cluster schedulers. Cluster schedulers [3, 65, 79, 84, 91, 100, 181] do not cater to the performance objectives of streaming video analytics. They take resource demands from tasks (not the profiles), mostly allocate based on fairness/priorities, and do not resize running containers, key to dealing with resource churn in VideoStorm (§2.6).

Deadline-Based Scheduling. Many systems [46, 79, 84, 114, 180] adaptively allocate resources to meet deadlines of batch jobs or reduce lag of streaming queries. Scheduling in real-time systems [110, 184] has also considered using utility functions to provide (soft) deadlines to running tasks. Crucially, these systems do not consider approximation together with resource allocation to meet deadlines and do not optimize across multiple queries and servers.

Streaming and Approximate Query Processing Systems. Load shedding has been a topic of interest in streaming systems [54, 144] to manage memory usage of SQL operators but they do not consider lag in processing. Aurora, Medusa, and Borealis [42, 69, 73] and follow-up works [170, 171, 175, 176, 187] use QoS graphs to capture lag and sampling rate but they consider them *separately* and do not trade-off between them, a key aspect in our solution. In contrast to JetStream [156], that degrades data quality based on WAN bandwidths, VideoStorm identifies the best knobs to use automatically and adjusts allocations *jointly* across queries. Stream processing systems used in production [1, 9, 131, 192] do not consider load-shedding, and resource-quality trade-off and lag in their design; Google Cloud Dataflow [44] requires manual trade-off specifications. Approximation is also used by recent [43, 48, 178] and older [96, 111] batch querying systems using statistical models for SQL operators [77].

Relative to the above literature, our main contributions are three-fold: (i) considering quality and lag of video queries *together* for multiple queries using *predictive control*, (ii) dealing with multitude of knobs in vision algorithms, and (iii) profiling *black-box vision transforms* with arbitrary user code (not standard operators).

Utility functions. Utility functions are used extensively throughout economics [139, 161], compute science [97, 113, 119, 133], and other disciplines to map how users benefit from performance [104, 120, 172]. In stream processing systems, queries describe their requirements for throughput, latency, and fraction of dropped tuples [46, 68, 122, 170]. With multiple entities, previous work has typically maximized the minimum utility [126, 136] or sum of utilities [126, 133], which is what we also use. Utility *elicitation* [60, 64, 70] helps obtain the exact shape of the utility function.

Autonomic Computing. Autonomic computing [52, 56, 61, 140, 151, 168] allocate resources to VMs and web applications to maximize their quality of service. While some of

them used look-ahead controllers based on MPC [142], they mostly ignored our main issues on the large space of configurations and quality-lag trade-offs.

2.9 Conclusion

VideoStorm is a video analytics system that scales to processing thousands of video streams in large clusters. Video analytics queries can adapt the quality of their results based on the resources allocated. The core aspect of VideoStorm is its scheduler that considers the resource-quality profiles of queries, each with a variety of knobs, and tolerance to lag in processing. Our scheduler optimizes jointly for the quality and lag of queries in allocating resources. VideoStorm also efficiently estimates the resource-quality profiles of queries. Deployment on an Azure cluster of 101 machines show that VideoStorm can significantly outperform a fair scheduling of resources, the widely-used policy in current clusters.

Chapter 3

SLAQ: Quality-Driven Scheduling for Distributed Machine Learning

Machine learning (ML) is an increasingly important tool for large-scale data analytics, including online search, marketing, healthcare, and information security. A key challenge in analyzing massive amounts of data with ML arises from the fact that model complexity and data volume is growing much faster than hardware speed improvements. Thus, time-sensitive machine learning on large datasets necessitates the use and efficient management of cluster resources. Three key features of ML are particularly relevant to resource management.

ML algorithms are intrinsically approximate. ML algorithms generally consist of two stages: *training* and *inference*. The training stage builds a model from a training dataset (e.g., images with labeled objects), and the inference stage uses the model to make predictions on new inputs (e.g., recognizing objects in a photo). ML models are intrinsically approximate functions for input-output mapping. We use *quality* to measure how well the model maps input to the correct output.

ML training is typically iterative with diminishing returns. While the inference stage is often lightweight and can run in real-time, the training stage is computationally expensive and usually requires multiple passes over large datasets. It generates a low-quality model at the beginning and improves the model's quality through a sequence of training iterations until it converges. In general, the quality improvement diminishes as more iterations are completed.

Training ML is an exploratory process. ML practitioners retrain their models repeatedly to explore feature validity [50], tune hyperparameters [107, 128, 134, 166], and adjust model structures [94] before they operationalize their final model, which is deployed for performing inference on individual inputs. The goal of retraining is to get the final model with the best quality. Since ML training jobs are expensive, practitioners in experimental environments often prefer to work with more approximate models trained within a short period of time for preliminary validation and testing, rather than wait a significant amount of time for a better trained model with poorly tuned configurations. In fact, algorithm tuning is an empirical process of trial and error that can take significant effort, both human and machine. With the exponential growth of data volume, the cost of decision making on model configurations will likely continue to increase.

Many ML frameworks have been developed [14, 18, 41, 138] to run large-scale training jobs in clusters with shared resources. Existing schedulers primarily focus on *resource fairness* [3, 19, 59, 87, 100, 106], but are agnostic to model quality and job runtime. During a burst of job submissions, equal resources will be allocated to jobs that are in their early stages and could benefit significantly from extra resources as those that have nearly converged and cannot improve much further. This is not efficient.

We present SLAQ, a cluster scheduling system for ML training jobs that aims to maximize the overall job quality. SLAQ dynamically allocates resources based on job resource demands, intermediate model quality, and the system's workload. The intuition behind

SLAQ is that in the context of approximate ML training, more resources should be allocated to jobs that have the most potential for quality improvement.

SLAQ leverages the fact that most ML training algorithms are implemented as an iterative optimization process. By continually monitoring the history of quality improvement and runtime, SLAQ generates highly-tailored and accurate quality predictions for future training iterations. SLAQ estimates the impact of resource allocation on model quality, and explores the quality-runtime trade-offs across multiple jobs. Based on this information, SLAQ adjusts their resource allocations of all running jobs to best utilize the limited cluster resources. The SLAQ scheduler is designed to be dynamic and fine-grained, so that resource allocations can adapt quickly to jobs' quality and the system's workload changes.

Challenges and solutions. In designing SLAQ, we had to overcome several technical challenges.

First, ML training algorithms measure the quality of models with tens of different metrics, which makes it difficult to compare the training progress of different jobs. SLAQ normalizes these metrics using the reduction of loss values. These intermediate quality measures are reported directly by the application APIs. Our normalization effectively unifies the quality measures for a broad set of ML algorithms.

Second, SLAQ should be able to precisely predict the impact that an extra unit of resources would have on the quality and runtime of ML training jobs. Previous work [179] predicts a job's runtime based on its computation and communication structure, but it requires that the job be analyzed or profiled offline. Unfortunately, the significant overhead of this offline analysis is prohibitive for our exploratory setting. SLAQ uses online prediction: it predicts the time and quality of the coming iterations based on statistics collected from previous iterations.

SLAQ supports configurable high-level goals when scheduling jobs. When maximizing the aggregate quality improvement, it can best utilize the cluster resources and achieve a

higher total quality gain across all jobs. When maximizing the minimum quality, SLAQ can achieve the equivalent of max-min fairness applied to quality (rather than resource allocation).

While we designed our scheduler for ML training applications, SLAQ can schedule many applications with approximate intermediate results. Some approximate jobs produce partial results at intermediate points of the application's run [194], while others generate approximate results from samples to avoid scanning entire datasets [43]. Improvement in the quality of these systems' results also diminishes with more processing time [200]. To that end, SLAQ's techniques are broadly applicable to other data analytics systems that employ iterative approximation approaches.

On the other hand, while SLAQ works with a large class of important ML algorithms, some non-convex ML algorithms are not currently supported. The convergence properties and optimization of these algorithms are being actively studied, and we leave scheduling support for these algorithms to future work.

We implemented SLAQ as a new scheduler within the Apache Spark framework [191]. SLAQ can use its quality-driven scheduling for many of the ML algorithms available in MLlib [138], Spark's machine learning package. In fact, SLAQ supports unmodified ML applications using existing MLlib optimizers, as well as applications using new optimization algorithms with only minor modifications. We evaluate various distinct ML training algorithms on datasets collected from various online sources. We found that SLAQ improves the average quality by up to 73% and reduces the average delay by up to 44% compared to fair resource scheduling.

3.1 Background and Motivation

The past several years has seen a rapid increase in both the volume of data used to train ML models and the size and complexity of these models. Growth in the performance of

the underlying hardware, however, has not caught up, thus placing higher demands on the computational resources used for this purpose.

An important way that data scientists cope with these demands is to leverage more approximate models for preliminary testing, in order to exclude bad trials and iterate to the right configuration. A significant amount of time and resource usage can potentially be saved because of the iterative nature of ML optimization algorithms, and the diminishing returns of quality improvements during the training iterations. Today's schedulers, however, do not provide a ready means to follow this strategy; a traditional max-min fair scheduler (similarly, the dominant resource fair scheduler [87]) ensures fair *resource* allocation without considering the potential of these resources to improve model quality.

This section motivates and provides background for SLAQ. §3.1.1 describes the iterative nature of the ML training process and how it is characterized by diminished returns. We introduce the exploratory training process in §3.1.2 and describe current practices in §3.1.3. We discuss the problems with existing cluster schedulers and propose our quality-aware scheduler in §3.1.4.

3.1.1 ML Training: Iterative Optimization Process

The algorithms used for the ML training process typically include a dataset specification, a loss function, an optimization procedure, and a model [90]. A machine learning model is a parametric transformation $f_\theta : X \mapsto Y$ that maps input variables to output variables, and it typically contains a set of parameters θ which will be regularly adjusted during the training process. The loss function represents how well the model maps training examples to correct output, and is often combined with a regularization term to incorporate generalizability. Training machine learning models can be summarized as optimizing the model parameters to minimize the loss function when applying the model on a dataset.

When the machine learning model is nonlinear, most loss functions can no longer be optimized in closed form. Algorithms such as Gradient Descent, L-BFGS and Expectation

Maximization (EM) are widely used in practice to *iteratively* solve the numerical optimization of the loss function. As the sizes of the dataset and model grow, the batch algorithms can no longer solve the optimization problem efficiently. Instead, various new algorithms have been proposed to improve the efficiency of the optimization process in an iterative and distributed fashion. For example, stochastic gradient descent (SGD) [62] reduces computationally complexity by evaluating the loss function and gradient on a randomly drawn subset of the overall dataset in each iteration.

The training process with the iterative optimization algorithms can be viewed as a refinement loop of the model. After initializing the parameter values (e.g., with random values), the optimization algorithms calculate changes on parameters in order to reduce the loss function, and update the model with new parameter values. This process continues until the decrease in the loss function falls below a certain threshold, or until a preset number of iterations have elapsed.

Another approach that some ML algorithms take is ensemble learning. Instead of training a complicated model with a large number of parameters, these algorithms focus on aggregating results from multiple diverse but small *submodels*. For example, boosting algorithms improve the accuracy of the model classifier by examining the errors in results, adding new submodels to the ensemble, and adjusting the weights of the set of submodels. Boost aggregating (bagging) algorithms train multiple submodels on different subsets of the training data by sampling with replacement. The training process of the ensemble models involves both iteratively refining each submodel, and iteratively adding new submodels or adjusting the weights of existing components.

When training a machine learning model, the first several iterations generally boost the quality very quickly. This is because the initial parameters of a model are generally set randomly. However, for most ML training algorithms, the quality improvements are subject to diminishing returns; iterations in later stages continue to cost the same amount of computational resources while making only marginal improvements on model quality as the results

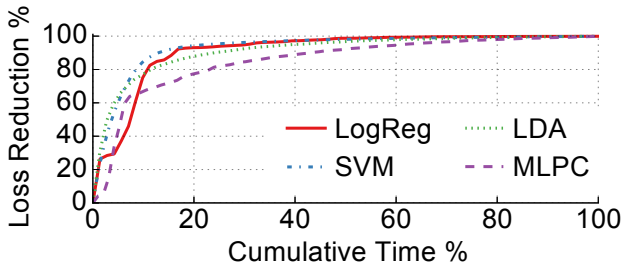


Figure 3.1: Cumulative time to achieve different percentages of loss reduction with four jobs: Logistic Regression (LogReg), Support Vector Machine (SVM), Latent Dirichlet Allocation (LDA) and Multi-Layer Perceptron Classifier (MLPC). Job convergence is defined to be $1/10000$ of initial loss reduction.

finally converge. For example, error in gradient descent algorithms on convex optimization problems often converges approximately as a geometric series [66]. Theoretically, at the kth iteration, the loss function reduction is $O(\mu^k)$, where μ is the convergence rate ($|\mu| < 1$). In general, loss reduction (quality improvement) diminishes as more iterations are completed.

Figure 3.1 plots the relative cumulative time to achieve different percentages of loss reduction. For example, it takes 20% time for the SVM job to reduce loss by 95%, and 80% time to further reduce it until convergence. Jobs for ML algorithm debugging and model tuning only require the training process to be almost completed to tell potentially good configurations from bad trials, and thus could save a lot of time and resources.

The law of diminishing returns applies in many other data analytics systems in addition to machine learning. Sampling-based approximate query processing systems compute approximate results by processing only a sample of the entire dataset in order to reduce resource usage and processing delay [43, 48, 53, 178]. Databases can also take advantage of online aggregation to incrementally refine the approximated results of SQL aggregate queries [96, 153, 194]. Using the *error* or *uncertainty* as a measurement of quality in these queries, we can observe that in most cases the convergence rate of these metrics are also monotonically decreasing.

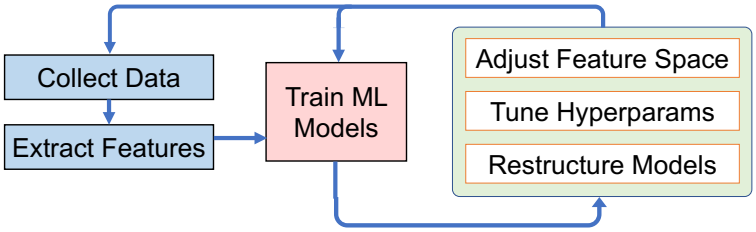


Figure 3.2: Retrain machine learning models.

3.1.2 Retraining Machine Learning Models

Training machine learning models is not a one-time effort. ML practitioners often train a model on the same dataset multiple times for exploratory purposes. This process provides early feedback to practitioners and helps direct their search for high quality models.

Feature engineering. Many ML algorithms require a featurized representation of the input data for efficient training and inference. For example, a speech recognition algorithm utilizes the discretized frequency features extracted from continuous sound signals with Fourier transforms and knowledge about the human ear [174]. Identifying exactly the useful features that yield the best quality relies on both domain knowledge and many training experiments.

Hyperparameter tuning. Many ML models expose *hyperparameters* that describe the high-level complexity or capacity of the models. Optimal values of these hyperparameters typically cannot be learned from the training data. Examples of hyperparameters include the number of hidden layers in a neural network, the number of clusters in a clustering algorithm, and the learning rate of mode parameters. It is desirable to explore different combinations of hyperparameter values, train multiple models, and use the one that gives the best result.

Model structure optimization. To ship ML models and run inference tasks on mobile and IoT devices, large models need to be compressed to reduce the energy consumption

and accelerate the computation. Various model compression techniques have been developed [94, 125]. These methods usually prune the unnecessary parameters of the model, retrain the model with the modified structure, and then prune again. This requires training the same job multiple times to get the best compression without compromising the quality of the model.

In addition, the interactions between features, hyperparameters and model structures make it even harder to search for the best model configuration. For example, features are often correlated with one another, and modifying the set of features also requires recalibrating the hyperparameters (such as learning rate). Expensive model configuration decisions demand highly efficient resource management in shared clusters.

3.1.3 Current Practices in ML Training

When exploring the ML model configuration space, users often submit training jobs with either a time cutoff or a loss value cutoff. Both monitoring heuristics are widely used in practice but have significant drawbacks.

Training ML models within a fixed time frame often results in unpredictable quality. This is because it is often difficult to predict a priori what the loss values will be at the deadline. More importantly, when a training job shares cluster resources with other jobs, the number of iterations completed by the deadline also depends on the cluster's workload and the scheduler's decisions.

A fixed loss (or fixed Δloss) cutoff is also difficult to reason about. Loss values in different algorithms are different in magnitude and have completely different meanings (further explained in §3.3.1). Additionally, with more complicated model structures and training algorithms, it is not rare to see the convergence rate of loss function fluctuate due to stochastic methods and model staleness [74]. Fixed loss values also make users lose the potential to gain further improvement on the training.

Some users choose to manually monitor the loss function values during the training process and stop the job when they think the models are good enough. However, large-scale ML jobs could take hours or even days to complete, which makes the monitoring impractical.

In the context of exploratory ML training, it is desirable to explore the quality-runtime trade-off across multiple concurrent jobs. SLAQ automates this process and obviates the need for the user to reason about arbitrary trade-offs. SLAQ flexibly fulfills a broad range of requirements for quality and delay of ML trainings, from approximate but timely models, to more traditional accurate model training. It allows users to stop jobs early before perfect convergence, and obtain a model with a loss function converged enough with much shorter latency.

3.1.4 Cluster Scheduling Systems

A cluster scheduler is responsible for managing resource allocation across multiple jobs. Modern data analytics frameworks (such as Hadoop [2], Spark [191], etc.) typically have two layers of scheduling: the job-level scheduler allocates resources to concurrent jobs running on the workers, while the task-level scheduler focuses on assigning tasks within a job to the available workers.

Existing job-level schedulers (Yarn [3], Mesos [100], Apollo [65], Hadoop Capacity [19], Quincy [106], etc.) mostly allocate resources based on resource fairness or priorities. For ML training jobs, however, these schedulers often make suboptimal scheduling decisions because they are agnostic to the progress (quality improvement) within each job. We argue that the scheduler should collect quality and delay information from each job and dynamically adjust the resource allocation to optimize for cluster-wide quality improvement.

SLAQ is a *fine-grained* job-level scheduler: it focuses on the allocation of cluster resources between competing ML *jobs*, but does so over *short time intervals* (i.e., hundreds

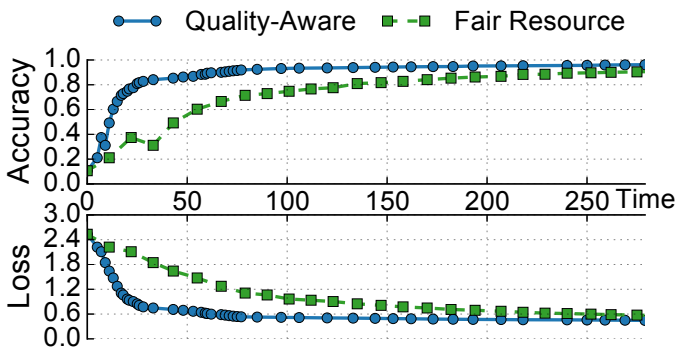


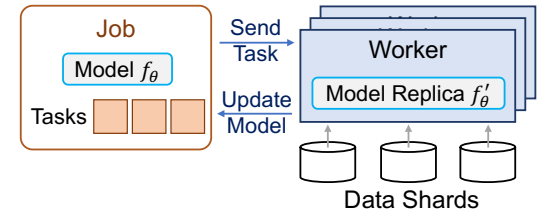
Figure 3.3: Accuracy (top) and loss function values (bottom) of a job with resources allocated by a quality-aware scheduler and a fair scheduler. Accuracy (percentage of correctly predicted data points) is evaluated on a testing dataset at the end of each training iteration. The more resources allocated to a job, the faster an iteration can be finished.

of milliseconds to a few seconds). Scheduling on short intervals ensures the continued rebalancing of resources across jobs, whose iteration time varies from tens to hundreds of milliseconds.

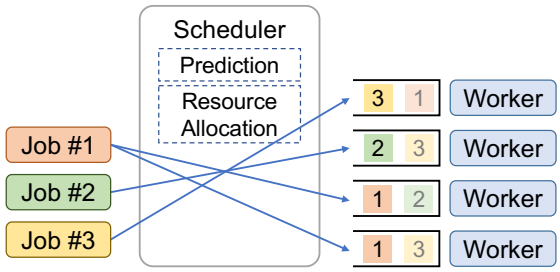
In a shared cluster with multiple users constantly submitting their training jobs, Figure 3.3 shows how the accuracy and loss values of one job change over time. With the fair scheduler, the job receives its fair share of cluster resources throughout its lifetime. A key observation here is that if we had given this job more resources in its early stages, its accuracy (loss) could have increased (decreased) much faster. SLAQ does exactly this, allocating more resources to the job when its potential improvement is large. In particular, the job was able to achieve 90% accuracy within a much shorter time frame (70s) with SLAQ than with the fair scheduler (230s). Especially for exploratory training jobs, this level of accuracy is frequently sufficient.

3.2 System Overview

SLAQ is a cluster management framework that hosts multi-tenant approximate ML training jobs running on shared resources. A centralized SLAQ scheduler coordinates the resource allocation of multiple ML training jobs. As shown in Figure 3.4(a), each job is composed



(a) Distributed ML Training



(b) Scheduler Architecture

Figure 3.4: Running ML training jobs with SLAQ.

of a set of tasks. Each task processes data based on the ML algorithm on a small partition of the dataset, and can be scheduled to run on any node. The driver program contains the iterative training logic, generates tasks for each iteration, and tracks the overall progress of the job. In the case of training ML models, a task generates an update to the model parameters based on a partition of the training dataset. The duration of a task typically ranges from tens of milliseconds to a few seconds. When the tasks finish processing the data, the updates from all tasks are aggregated and sent back to the job driver program to update the primary copy of the model.

Similar to many cluster management systems, SLAQ divides machines into smaller *workers*, which is the minimum unit of resource to run a task. Figure 3.4(b) shows that each job driver, at a certain time, can send tasks to the workers allocated to that job in the cluster.

The SLAQ scheduler directly communicates with the drivers of currently running jobs to track their progress and update their resource allocation periodically. At the beginning of each scheduling epoch, SLAQ allocates resources between all the jobs based on system workload, the demands, and progress of the jobs. The scheduler reclaims workers back from some job drivers, and reallocates them to other jobs for better system-wide performance

goals. Note that this is very different from many of the existing cluster managers [3, 19] which only statically allocate resources to jobs before they get started.

We made this decision because of two reasons. First, unlike general batch processing, jobs that train ML models are typically iterative and usually need longer time to complete. Scheduling only at the start of the job is too coarse-grained and can easily lead to starvation or underutilization of system resources. Second, the quality improvement of the training jobs often changes rapidly (as described in §3.1.1). Fixed allocation makes the scheduler unable to adapt to jobs' changes in quality improvement and resource demands.

3.3 Design

This section describes the mechanisms by which SLAQ addresses its key challenges. First, how to normalize quality measures between distinct jobs in order to determine how quickly they are increasing (or not) in quality relative to one another (§3.3.1). Second, how SLAQ uses jobs' resource usage and quality information to precisely predict the impact of resource allocation in an online fashion (§3.3.2). Third, how SLAQ allocates resources to maximize system-wide quality improvement (§3.3.3).

3.3.1 Normalizing Quality Metrics

As explained in §3.1.1, ML training algorithms are designed to be an optimization process which iteratively minimizes a loss function, and thus improves the model's quality. ML algorithms use various different measurement metrics to indicate the quality of model training. Though comparing a single job's quality improvement across iterations is simple, comparing these metrics across different jobs presents a challenge. To schedule for better overall quality, we need to compare the quality metrics across different jobs. This enables SLAQ to trade off resources and quality between jobs.

Algorithm	Type	Optimization Algorithm	Dataset
K-Means	Clustering	Lloyd Algorithm	Synthetic
LogReg	Classification	Gradient Descent	Epsilon [32]
SVM	Classification	Gradient Descent	Epsilon
SVMPoly	Classification	Gradient Descent	MNIST [27]
GBT	Classification	Gradient Boosting	Epsilon
GBTReg	Regression	Gradient Boosting	YearPredictionMSD [26]
MLPC	Classification	L-BFGS	Epsilon
LDA	Clustering	EM / Online Algorithm	Associated Press Corpus [11]
LinReg	Regression	L-BFGS	YearPredictionMSD

Table 3.1: Summary of ML algorithms, types, and the optimizers and datasets we used for testing. The algorithms include K-Means, Logistic Regression (LogReg), Support Vector Machine (SVM), SVM with polynomial kernel (SVMPoly), Gradient Boosted Tree (GBT), GBT Regression (GBTReg), Multi-Layer Perceptron Classifier (MLPC), Latent Dirichlet Allocation (LDA), and Linear Regression (LinReg).

One straightforward solution is to use a universal metric such as *accuracy* to measure the model quality. Accuracy represents the percentage of correctly predicted data points, and the range is always from 0 to 1. Similarly, the F1 score, ROC curve, and confusion matrix also measure the model quality taking the false positive and false negative ratios and multi-class results into consideration [154]. While these metrics are intuitively understandable to classification algorithms, they are not applicable to non-classification algorithms such as regression or unsupervised learning. In addition, accuracy and similar metrics require constructing a model and evaluating that model against a labeled validation set, which introduces an additional overhead to the job.[1]

Loss normalization. In contrast to the accuracy metrics, the loss function is calculated by the algorithm itself in each iteration, incurring no additional overhead. However, each algorithm's loss function has a different real-world interpretation. The range, convexity, and monotonicity of the loss functions depend on both the models and the optimization algorithms [90]. Directly normalizing loss values requires a priori knowledge of the loss range, which is impractical in an online setting.

[1] Validation is commonly used in ML training to prevent overfitting. Due to the overhead, however, model evaluation on the validation set is usually performed once every several iterations, not every iteration.

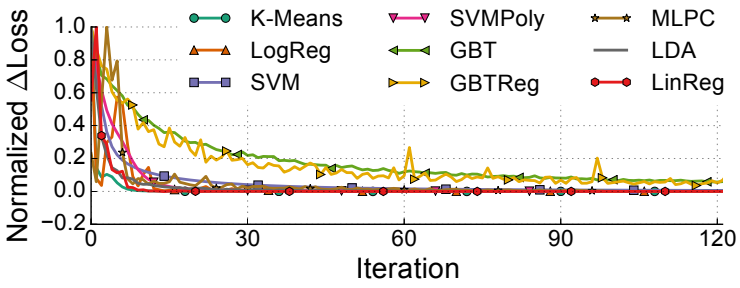


Figure 3.5: Normalized ΔLoss for ML algorithms.

For example, clustering algorithms (e.g., K-Means) use the *sum of squared distances to the cluster centroids* as the loss function. Classification and regression algorithms (e.g., SVM, Linear Regression, etc.) commonly use hinge or logistic gradient loss which represents *discrepancy of prediction* on the training data. The range of the measured values can vary by orders of magnitude: K-Means on our synthetic dataset reduces the loss from 300 down to 0, and the range highly depends on the absolute coordinates of the data points; on the other hand, SVM on a handwritten digit recognition dataset [27] reduces the loss from 1 down to 0.4. Unfortunately, there are no known analytical models to predict these ranges without actually running the training jobs.

Based on the convergence properties of loss functions (further explained in §3.3.2), we choose to normalize the *change* in loss values between iterations, as opposed to the loss values themselves. Most optimizers used in training algorithms try to reduce the values of loss functions, and for convex optimization problems, the values decrease monotonically [66]. The convergence rate, because of the diminishing returns, generally decreases in later iterations. So for a certain job, we normalize the change of loss values in the current iteration with respect to the largest change we have seen so far.

Figure 3.5 shows the normalized changes of loss values for common ML algorithms (summarized in Table 3.1). Because a loss function eventually converges to a certain value, the corresponding change of loss values always converges to 0. As a result, even though the set of algorithms have diverse loss ranges, we observe that they generally follow similar

convergence properties, and can be normalized to decrease from 1 to 0. This helps SLAQ track the progress of different training jobs, and, for each job, correctly project the time to reach a certain loss reduction with a given resource allocation.

SLAQ supports a large class of important ML algorithms, but currently does not support some non-convex optimization algorithms due to the lack of convergence analytical models.

3.3.2 Measuring and Predicting Loss

After unifying the quality metrics for different jobs, we proceed to allocate resources for global quality improvement. When making a scheduling decision for a given job, SLAQ needs to know how much loss reduction the job would achieve by the next epoch if it was granted a certain amount of resources. We derive this information by predicting (i) how many iterations the job will have completed by the next epoch (§3.3.2), and (ii) how much progress (i.e., loss reduction) the job could make within these iterations (§3.3.2).

Prediction for iterative ML training jobs is different from general big-data analytics jobs. Previous work [45, 179] estimates job's runtime on some given cluster resources by analyzing the job computation and communication structure, using offline analysis or code profiling. As the computation and communication pattern changes during ML model configuration tuning, the process of offline analysis needs to be performed every time, thus incurring significant overhead. ML prediction is also different from the estimations to approximate analytical SQL queries [43, 194] where the resulting accuracy can be directly inferred with the sampling rate and analytics being performed. For iterative ML training jobs, we need to make *online* predictions for the runtime and intermediate quality changes for each iteration.

Runtime Prediction

SLAQ is designed to work with distributed ML training jobs running on batch-processing computational frameworks like Spark and MapReduce. The underlying frameworks help

achieve *data parallelization* for training ML models: the training dataset is large and gets partitioned on multiple worker nodes, and the size of models (i.e., set of parameters) is comparably much smaller. The model parameters are updated by the workers, aggregated in the job driver, and disseminated back to the workers in the next iteration.

SLAQ's fine-grained scheduler resizes the set of workers for ML jobs frequently, and we need to predict the runtime of each job's iteration, even while the number and set of workers available to that job is dynamically changing. Fortunately, the runtime of ML training—at least for the set of ML algorithms and model sizes on which we focus—is dominated by the computation on the partitioned datasets. SLAQ considers the total CPU time of running each iteration as $c \cdot S$, where c is a constant determined by the algorithm complexity, and S is the size of data processed in an iteration. SLAQ collects the aggregate worker CPU time and data size information from the job driver, and it is easy to learn the constant c from a history of past iterations. SLAQ thus predicts an iteration's runtime simply by $c \cdot S/N$, where N is the number of worker CPUs allocated to the job.

We use this heuristic for its simplicity and accuracy (validated through evaluation in §3.5.3), with the assumption that communicating updates and synchronizing models does not become a bottleneck. Even with models larger than hundreds of MBs (e.g., Deep Neural Networks), many ML frameworks could significantly reduce the network traffic with model parallelism [129] or by training with relaxed model consistency with bounded staleness [78], as discussed in §3.6. Advanced runtime prediction models [167] can also be plugged into SLAQ.

Loss Prediction

Iterations in some ML jobs may be on the order of 10s–100s of milliseconds, while SLAQ only schedules on the order of 100s of milliseconds to a few seconds. Performing scheduling on smaller intervals would be disproportionally expensive due to scheduling overhead and lack of meaningful quality changes. Further, as disparate jobs have different iteration

periods, and these periods are not aligned, it does not make sense to try to schedule at "every" iteration of the jobs.

Instead, with runtime prediction, SLAQ knows how many iterations a job could complete in the given scheduling epoch. To understand how much quality improvement the job could get, we also need to predict the loss reduction in the following several iterations.

A strawman solution is to directly use the loss reduction obtained from the last iteration as the predicted loss reduction value for the following several iterations. This method actually works reasonably well if we only need to predict one or two iterations. However, this could perform poorly in practice when the number of iterations per scheduling epoch is higher. This could be the case, for example, when the training dataset is small or an abundance of resources is allocated to the job.

We can improve the prediction accuracy by leveraging the convergence properties of the loss functions of different algorithms. Based on the optimizers used for minimizing the loss function, we can broadly categorize the algorithms by their convergence rate.

Algorithms with sublinear convergence rate. First-order algorithms in this category[2] have a convergence rate of $O(1/k)$, where k is the number of iterations [95]. For example, gradient descent is a first-order optimization method which is well-suited for large-scale and distributed computation. It can be used for SVM, Logistic Regression, K-Means, and many other commonly used machine learning algorithms. With optimized versions of gradient descent, the convergence rate could be improved to $O(1/k^2)$.

Algorithms with linear or superlinear convergence rates. Algorithms in this category[3] have a convergence rate of $O(\mu^k), |\mu| < 1$. For example, L-BFGS, which is a widely used Quasi-Newton Method, has a superlinear convergence rate which is between linear and quadratic. It can be used for SVM, Neural Networks, and others.

[2] Assume the loss function f is convex, differentiable, and ∇f is Lipschitz continuous.

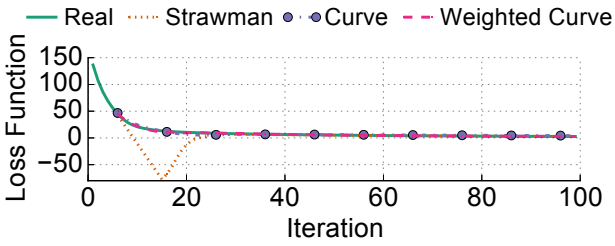
[3] Assume the loss function f is convex and twice continuously differentiable, optimization algorithms can take advantage of the second-order derivative to get faster convergence.

Distributed optimization algorithms. Optimization algorithms like gradient descent require a full pass through the complete dataset to update the model's parameters. This can be very expensive for large jobs which have data partitions stored on multiple nodes. Distributed ML training benefits from stochastic optimization algorithms. For example, stochastic gradient descent (SGD) processes a mini-batch (samples extracted from a subset of the training data) at a time and updates the parameters in each step. The significant efficiency improvement of SGD comes at the cost of slower convergence and fluctuation in loss functions. In terms of number of iterations, however, SGD still converges at a rate of $O(1/k)$ with properly randomized mini-batches.

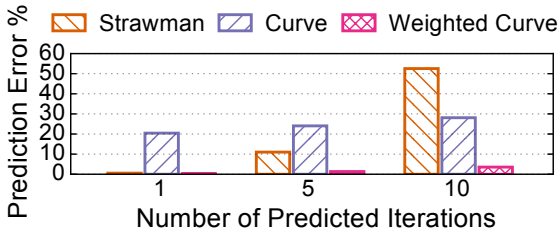
With the assumptions of loss convergence rate, we use curve fitting to predict future loss reduction based on the history of loss values. For the set of machine learning algorithms we consider, we use the history of loss values at a certain time to fit a curve $f(k) = \frac{1}{ak^2+bk+c} + d$ for sublinear algorithms, or $f(k) = \mu^{k-b} + c$ for linear and superlinear algorithms.

We further improve the prediction accuracy using exponentially weighted loss values. Intuitively, loss values obtained in the near past are more informative for predicting the loss values in the near future. The weights assigned to loss values decay exponentially when new iterations finish, and the parameters of the curve equations get adjusted for each prediction.

Figure 3.6 shows the loss values predicted using the different methods described above. The strawman solution works well when predicting only one iteration in advance, but degrades quickly as the number of iterations to predict increases. The latter scenario is likely because SLAQ makes a scheduling decision once every epoch, which typically spans multiple iterations. In contrast, as shown in Figure 3.6(b), the weighted curve fitting method achieves a low average prediction error of 3.5% even when predicting up to 10 iterations in advance.



(a) Predicting loss values 10 iterations in advance.



(b) Average loss prediction errors when predicting 1, 5 and 10 iterations in advance.

Figure 3.6: Predicting loss values with 3 methods.

3.3.3 Scheduling Based on Quality Improvements

With accurate runtime and loss prediction, SLAQ allocates cluster CPUs to maximize the system-wide quality. SLAQ can flexibly support different optimization metrics, including both maximizing the total (sum) quality of all jobs, as well as maximizing the minimum quality (equivalent to max-min fairness) across jobs.

Maximizing the total quality. We schedule a set of J jobs running concurrently on the shared cluster for a fixed scheduling epoch T, i.e., a new scheduling decision can only be made after time T. The optimization problem for maximizing the total normalized loss reduction over a short time horizon T is as follows. Sum of allocated resources a_j cannot exceed the cluster resource capacity C.

$$\begin{aligned} \max_{j \in J} \quad & \textstyle\sum_j Loss_j(a_j, t) - Loss_j(a_j, t+T) \\ s.t. \quad & \textstyle\sum_j a_j \leq C \end{aligned}$$

Algorithm 1 Maximizing Total Loss Reduction

- *epoch*: scheduling time epoch
- *num_cores*: total number of cores available
- *alloc*: number of cores allocated to jobs
- *prior_q*: priority queue containing jobs and their loss reduction values if allocated with one extra core

```
1: function PREDICTLOSSREDUCTION(job)
2:     pred_loss = PREDICTLOSS(job, alloc[job], epoch)
3:     pred_loss_p1 = PREDICTLOSS(job, alloc[job] + 1, epoch)
4:     return pred_loss − pred_loss_p1
5: function ALLOCATERESOURCES(jobs)
6:     for all job in active jobs do
7:         alloc[job] = 1
8:         num_cores = num_cores − 1
9:         pred_loss_red = PREDICTLOSSREDUCTION(job)
10:        prior_q.enqueue(job, pred_loss_red)
11:    while num_cores > 0 do
12:        job = prior_q.dequeue()
13:        alloc[job] = alloc[job] + 1
14:        num_cores = num_cores − 1
15:        pred_loss_red = PREDICTLOSSREDUCTION(job)
16:        prior_q.enqueue(job, pred_loss_red)
17:    return alloc
```

When including job j at allocation a_j, we are paying cost of a_j and receiving value of $\Delta l_j = Loss_j(a_j,t) - Loss_j(a_j,t+T)$. The scheduler prefers jobs with highest value of $\Delta l_j/a_j$; i.e., we want to receive the largest gain in loss reduction normalized by resource spent.

Algorithm 1 shows the resource allocation logic of SLAQ. We start with $a_j = 1$ for each job to prevent starvation. At each step we consider increasing a_i (for all queries i) by one unit (in our implementation, one CPU core) and use our runtime and loss prediction logic to get the predicted loss reduction. Among these queries, we pick the job j that gives the most loss reduction, and increase a_j by one unit. We repeat this until we run out of available resources to schedule.

Maximizing the total loss reduction targets the cost-effectiveness of cluster resources. This is desirable not only on clusters used by a single company which may have high resource contention, but potentially even on multi-tenant clusters (clouds) in which revenue could be directly associated with the total quality progress (loss reduction) of ML jobs.

Maximizing the minimum quality. Below is the optimization problem to minimize the maximum loss value (or equivalently, maximizing the minimum quality) over time horizon T. With a set of J jobs running concurrently, this scheduling policy makes sure no one is *falling behind*. We require that all loss values be no bigger than l and we minimize l.

$$\begin{aligned} \min_{j \in J} \quad & l \\ s.t. \quad & \forall j : Loss_j(a_j, t+T) \leq l \\ & \textstyle\sum_j a_j \leq C \end{aligned}$$

The system quality, in this case, is represented by the loss value l of the worst job j. The only way we can improve it is to reduce the loss value of j. Our heuristic is thus as follows. We start with $a_j = 1$, and at each step we pick job $i = \arg\min_j Loss_j(a_j, t+T)$. We increase its allocation a_i by one unit, recompute $Loss_i(a_i, t+T)$, and repeat this process until we run out of resources.

Maximizing the minimum quality achieves max-min fairness in model quality. It is especially useful for ML applications that include multiple collaborative models, and the overall quality is determined by the lowest quality of all the submodels. For example, a security application for network intrusion detection should train multiple collaborative models identifying distinct attacking patterns with max-min fairness in quality.

Prioritize jobs on shared clusters. The above scheduling policies are based on the assumptions that all the concurrently running jobs have equal importance, and thus they will be treated equally when comparing their quality. This can be easily adjusted to account for jobs with different importance by adding a weight multiplier to the jobs, identically to how max-min fairness can be easily changed to weighted max-min fairness.

For example, a cluster may host experiment jobs and production jobs for ML training, and a higher weight should be assigned to jobs for production uses. With the same training progress, a job with a higher weight will get its loss reduction proportionally amplified

by the scheduler compared to a normal job. Thus, high-priority jobs generally get more iterations finished with SLAQ.

Mixing ML with other types of jobs. SLAQ can also run non-ML jobs sharing the same cluster with approximate ML jobs. For non-ML jobs, the scheduler falls back to fairness or reservation based resource allocation. This effectively reduces the total capacity C available to all approximate ML jobs. SLAQ follows the same algorithms to maximize the total or minimum quality under varying resource capacity C.

3.4 Implementation

We implemented SLAQ within the popular Apache Spark framework [191], and utilize its accompaning MLlib machine learning library [138]. Spark MLlib describes ML workflow as a pipeline of *transformers*, and it provides a set of high-level APIs to help design ML algorithms on large datasets. Many commonly used ML algorithms are pre-built in MLlib, including feature extraction, classification, regression, clustering, collaborative filtering, and so on. These algorithms can easily be extended and modified for specific use cases.

The SLAQ prototype is implemented based on the Spark job scheduler. Multiple jobs place the ready tasks into task pools, which are then controlled and dispatched by SLAQ scheduler. The driver programs of ML jobs continually report their loss value information for each iteration they finish.

Token bucket. SLAQ uses a token bucket algorithm to implement the resource allocation policies described in §3.3.3. At each scheduling epoch, CPU time of all allocated cores is added to each job as *tokens*. SLAQ assigns tasks to available workers, and keeps track of how many tokens are consumed by those tasks by collecting Spark worker statistics. Tasks are throttled if the corresponding job has used up its tokens.

Running unmodified ML applications. ML *applications* written using Spark MLlib can directly run on SLAQ without any modifications. This is because SLAQ extends the underlying *optimizers* (e.g., SGD, L-BFGS, etc.) APIs to report loss values at each iteration. We cover most library algorithms provided in MLlib. Even when it is necessary to add new library algorithms, one can easily adopt SLAQ by reporting loss values using SLAQ's API. This is a one-line modification in most of the algorithms present in MLlib.

3.5 Evaluation

In this section, we present evaluation results on SLAQ. We demonstrate that SLAQ (i) provides significant improvement on quality and runtime for approximate ML training jobs, (ii) is broadly applicable to a wide range of ML algorithms, and (iii) scales to run a large number of ML training algorithms on clusters.

3.5.1 Methodology

Testbed. Our testbed consists of a cluster of 20 instances of c3.8xlarge machines on Amazon EC2 Cloud. Each worker machine has 32 vCPUs (Intel Xeon E5-2680 v2 @ 2.80 GHz), 60GB RAM, and is connected with 10Gb Ethernet links.

Workload. We tested our system with the most common ML algorithms derived from MLlib with minor changes, including (i) classification algorithms: SVM, Neural Network (MLPC), Logistic Regression, GBT, and our extension to Spark MLlib with SVM polynomial kernels; (ii) regression algorithms: Linear Regression, GBT Regression; (iii) unsupervised learning algorithms: K-Means clustering, LDA. Each algorithm is further diversified to construct different models. For example, SVM with different kernels, and MLPC Neural Network with different numbers of hidden layers and perceptrons.

Datasets. With the algorithms, our models are trained on multiple datasets we collected from various online sources with modifications, as well as on our synthetic datasets. The datasets span a variety of types (plain texts [11], images [27], audio meta features [26], and so on [21]). The size of the distinct datasets we use in each run is more than 200GB. In the experiments, all the training datasets are cached as Spark Dataframes in cluster shared memory. We set the fraction of data sample processed at each iteration to be 100%, i.e., the entire training data is processed in every iteration.

Baseline. The baseline we compare against is a work-conserving fair scheduler. It is the widely-used scheduling policy for cluster computing frameworks [3, 19, 87, 100, 106]. The fair scheduler evenly divides available resources to all active jobs. It also dynamically adjusts resource allocations to fair share when new jobs join and old jobs leave the system.

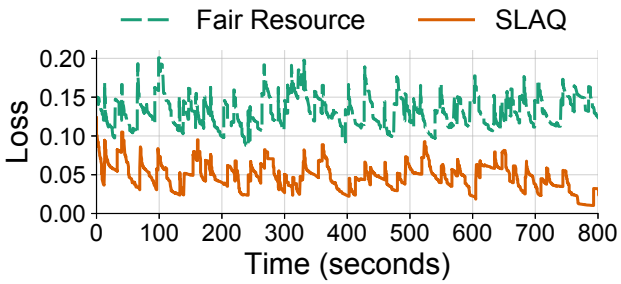
3.5.2 System Performance

Scheduler Quality and Runtime Improvement

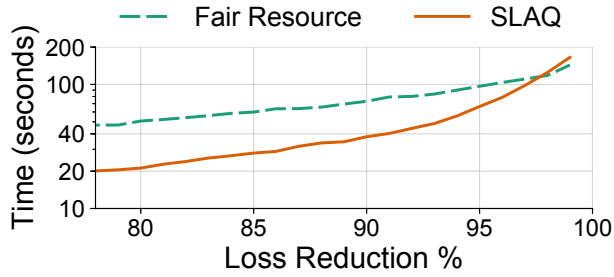
To evaluate job quality improvement, we first run a set of 160 ML training jobs with different algorithms, model sizes, and datasets on the shared cluster of 20 nodes. In the experiment, jobs are submitted to the cluster with their arrival time following a Poisson distribution (mean arrival time 15s). A job is considered fully converged when its normalized loss reduction is below a very small value, in this case, the loss reduction at the 100th iteration.[4] We compare the aggregate quality and runtime of these jobs between SLAQ and the fair scheduler.

Figure 3.7(a) shows the average normalized loss values across running jobs with SLAQ and the fair scheduler in an 800s time window of the experiment. When a new job arrives, its initial loss is 1.0, raising the average loss value of the active jobs; the spikes in the figure

[4]Recall that the loss reduction for each iteration is independent of the amount of resources the job is allocated; the resource allocation instead dictates the amount of *wall-clock time* each iteration takes.



(a) Average of normalized loss values.



(b) Time to achieve loss reduction percentage.

Figure 3.7: Comparing loss improvement and runtime between SLAQ and fair scheduler.

indicate new job arrivals. Yet because SLAQ allocates resources to maximize the total quality improvement (loss reduction), the average loss value of all active jobs using SLAQ is much lower than with the fair scheduler. In particular, SLAQ's average loss value is 0.49 at each scheduling epoch, which is 73% lower than that of the fair scheduler.

Figure 3.7(b) shows the average time it takes a job to achieve different loss values. As SLAQ allocates more resources to jobs that have the most potential for quality improvement, it reduces the average time to reach 90% (95%) loss reduction from 71s (98s) down to 39s (68s), 45% (30%) lower. At the very end of the job execution, further iterations take longer time as the job quality is less likely to be improved. Thus, in an environment where users submit exploratory ML training jobs, SLAQ could substantially reduce users' wait times.

Figure 3.8 explains SLAQ's benefits by plotting the allocation of CPU cores in the cluster over time. Here we group the active jobs at each scheduling epoch by their normalized loss: (i) 25% jobs with high loss values; (ii) 25% jobs with medium loss values; (iii) 50% jobs with low loss values (almost converged). With a fair scheduler, the cluster CPUs should

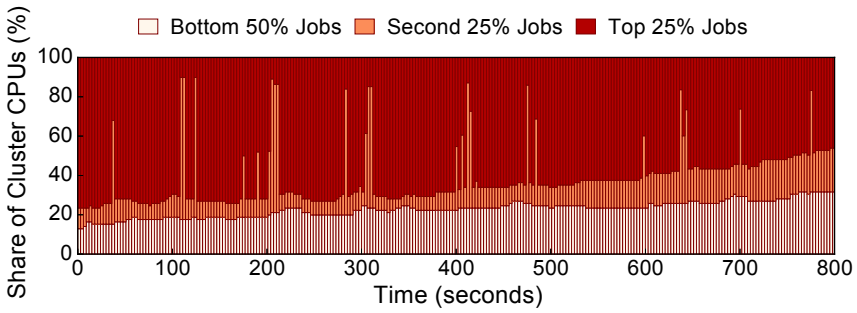


Figure 3.8: Resource allocation across jobs. At the beginning, jobs with the greatest 25% loss allocated vast majority of resources; towards the end, the difference in loss shrinks, the allocation is more spread out.

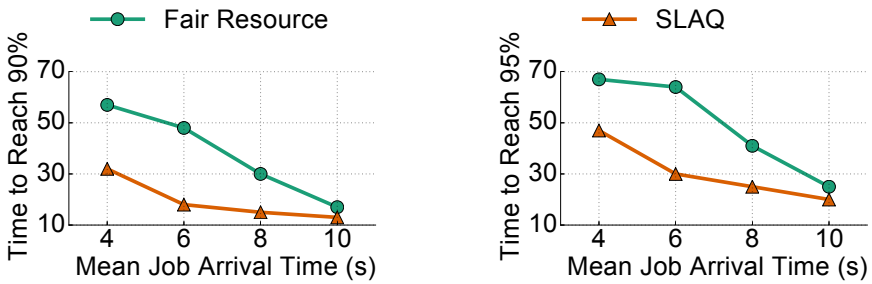


Figure 3.9: The performance difference between SLAQ and a fair resource scheduler is more significant under workloads with greater contention, e.g., jobs arriving with a mean arrival time of 4s compared to 10s.

be allocated to the three groups proportionally to the number of jobs. In contrast, SLAQ adapts to the job quality improvement, and allocates much more computation resource to (i) and (ii). In fact, jobs in group (i) take 60% of cluster CPUs, while jobs in group (iii), despite having 50% of the population, get only 22% of cluster CPUs on average. SLAQ transfers many resources from nearly converged jobs to the jobs that have the most potential for significant quality improvement, which is the underlying reason for the improvement in Figure 3.7.

Handling Different Workloads

The achieved qualities of training jobs strongly depend on the cluster workload. As the workload increases, it becomes more important to efficiently utilize the resources. In this

experiment, we vary the mean arrival time of new jobs, which in turn varies the number of concurrent jobs, and observe how SLAQ and the fair scheduler handle resource contention under different workloads.

Figure 3.9 illustrates that SLAQ achieves a greater relative benefit over a fair schedule under more contentious or aggressive workloads. We start with a mean arrival time of 10s (or equivalently, 6 new jobs per minute). Under the light workload, the computation resources are relatively abundant for each job, so the time to reach 90% (95%) loss reduction is similar for both schedulers, with SLAQ performing 23% (20%) better.

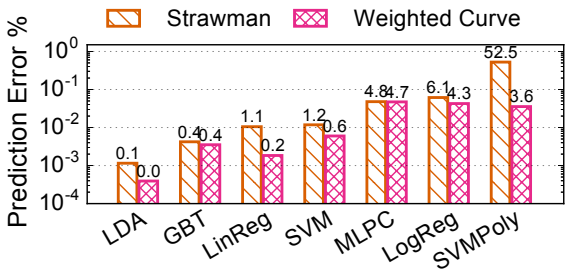
As we increase the system workload with smaller mean job arrival times, cluster resource contention increases. SLAQ allocates resources to the jobs with the greatest potential. As a result, when the mean arrival time is 4s (15 new jobs per minute), SLAQ achieves an average time for jobs to reach 90% (95%) loss reduction that is 44% (30%) less than the fair scheduler.

3.5.3 Robustness of Prediction

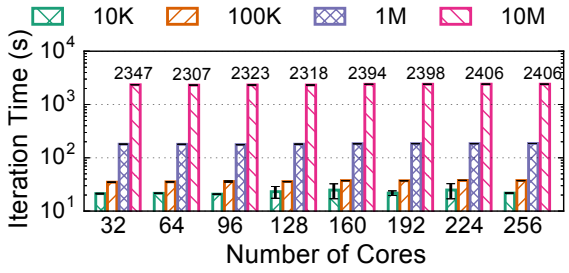
SLAQ relies on an estimate of the expected loss reduction of a job, given a certain resource alloction (see §3.3.2). To ensure stability, SLAQ makes a reallocation decision only once per scheduling epoch. Thus, the scheduler requires (i) the loss predictor to precisely estimate the loss values at least a few iterations in advance, and (ii) the runtime predictor to accurately report how long each iteration takes with a certain number of allocated cores.

Figure 3.10(a) plots the loss prediction error for the types of ML algorithms we tested (Table 3.1). We compare the loss prediction error relative to the true values for 10 iterations, with both strawman and weighted curve fitting methods of §3.3.2. Our prediction achieves less than 5% prediction errors for all the algorithms.

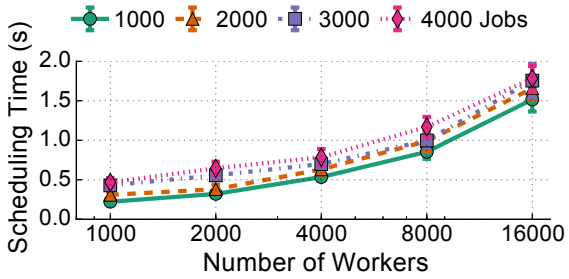
Recall that SLAQ uses a simple heuristic to estimate the iteration runtime with N cores. To demonstrate that each iteration's CPU time is $c \cdot S$ (c as a constant), regardless of how many workers are allocated, we evaluate the total CPU time to complete an iteration with



(a) Predicting the next 10th iteration.



(b) Average CPU time to finish each iteration.



(c) Scheduling time.

Figure 3.10: SLAQ loss / runtime prediction and overhead.

a fixed data size S. We vary the number of workers (32 cores each) between 1 and 8 and training neural network models of sizes from 10KB to 10MB. Figure 3.10(b) illustrates that, at least for ML models smaller than tens of MB, communication and model synchronization do not affect processing time. Therefore, when dynamically changing N, an iteration's time can simply be estimated as $c \cdot S/N$. We discuss extending SLAQ to large models in §3.6.

3.5.4 Scalability and Efficiency

Figure 3.10(c) plots the time taken by SLAQ to schedule tens of thousands of concurrent jobs on large clusters (simulating both the jobs and worker nodes). SLAQ makes its scheduling decisions in between hundreds of milliseconds to a few seconds, even when scheduling 4000 jobs across 16K worker cores. These decisions are made each scheduling epoch, a timeframe of a few seconds. As shown in Figure 3.6, the more iterations in advance SLAQ predicts, the larger potential error it will incur. The agility of SLAQ enables the scheduler to predict only a few iterations in advance for each ML training job, adjusting its resource allocation decisions frequenty to meet the jobs' quality goals. SLAQ's scheduling time is comparable to the scalability of schedulers in many big data clusters today, leading us to conclude that SLAQ is sufficiently fast and scalable for (rather aggressive) real-world needs.

3.6 Discussion

Communication overhead. SLAQ is tested with ML models that have a moderate number of parameters. Recent developments in distributed frameworks for training ML models, especially deep neural networks (DNN), incur more communication and synchronization overhead between the ML job driver and worker nodes. For example, with a large number of perceptrons and multiple layers, a DNN model can grow to tens of GBs [124, 146].

Since our current implementation is based on Spark, the driver essentially becomes a single-node parameter server [10], which is responsible for gathering, aggregating, and distributing the models in every iteration. This communication overhead—due to Spark's architecture—limits our ability to train large models.

Several solutions have been proposed to mitigate the communication overhead problem. Model parallelization using architectures based on parameter servers or graph computing proportionally scale the model serving nodes with the workers [14, 41, 129, 189]. With these

optimized frameworks, SLAQ's performance improvement based on online prediction and scheduling heuristics should apply to large ML models.

Distributed ML training with relaxed consistency. Distributed ML frameworks used in practice leverage a relaxed consistency model with bounded staleness [78] to reduce the communication costs during model synchronization. The convergence progress of the underlying ML training algorithms is typically robust to a certain degree of fluctuation and slack, so the efficiency improvement obtained from the parallelism outweighs the staleness slowdown on convergence rate.

A commonly used execution model with bounded staleness is Bulk Synchronous Parallel (BSP), which allows multiple workers to individually update on partitioned training data and only synchronizes their models every several iterations [74, 143, 189]. We can extend SLAQ to support these frameworks by collecting the batch iteration time on each worker, and the model quality and communication time at each synchronization barrier to help estimate the loss reduction under the two levels of iterativeness. In fact, the convergence property of ML training is also studied in [143] with the BSP execution model under various conditions (e.g., varying communication latency and cluster sizes).

Non-convex optimization. SLAQ's loss prediction is based on the convergence property of the underlying optimizers and curve fitting with the loss history. Loss functions of non-convex optimization problems are not guaranteed to converge to global minima, nor do they necessarily decrease monotonically. The lack of an analytical model of the convergence properties interferes with our prediction mechanism, causing SLAQ to underestimate or overestimate the potential loss reduction.

One solution to this problem is to let users provide the scheduler with hint of their target loss or performance, which could be acquired from state-of-the-art results on similar problems or previous training trials. The convergence properties and optimization of non-

convex algorithms is being actively studied in the ML research community [63, 123]. We leave modeling the convergence of these algorithms to future work.

3.7 Related Work on Scheduling ML Systems

Approximate computing systems. Many systems [43, 48, 53, 96, 111, 178] allow users to get approximate results with significantly reduced job completion time. Online aggregation databases [96, 194] generate approximate results and iteratively refine the quality. While we designed SLAQ for iterative ML training jobs, our techniques are broadly applicable to scheduling data analytics systems that iteratively refine their results.

Scheduling ML systems. Large-scale ML frameworks [14, 33, 41, 72, 129, 138, 163] optimize the computation and resource allocation for multi-dimensional matrix operators within a training job. These systems greatly accelerate the training process and reduce job's synchronization overhead. As a cluster scheduler, SLAQ could support different underlying ML frameworks (with modifications) in the future, and allocate resources at the job level to optimize across different ML training jobs.

ML model search. Several systems [166,167] are designed to accelerate the model searching procedure. TuPAQ [167] uses a planning algorithm to discover hyperparameter settings and exclude bad trials automatically. SLAQ is designed for ML training in general exploratory settings on multi-tenant clusters. Automated model search systems could work in conjuction with SLAQ for faster decisions and better cluster utilization.

Cluster scheduling systems. Existing cluster schedulers [3, 19, 59, 87, 100, 106] primarily focus on resource fairness, job priorities, cluster utilization, or resource reservations, but do not take job quality into consideration. They mostly ignore the quality-time trade-off, and

the quality trade-off between jobs. This trade-off space is crucial for ML training jobs to get approximate results with much less resource usage and lower latency.

Estimation of resource usage and runtime. Ernest [179] predicts job quality and runtime based on the internal computation and communication structures of large-scale data analytics jobs. CherryPick [45] improves cloud configuration selection process using Bayesian Optimization. Despite the generality, these systems require jobs to be analyzed offline. When users debug and adjust their models, the computation structure is likely to change very often, and thus the offline analysis will bring significant overhead. NearestFit [76] provides a progress indicator for a broad class of MapReduce applications with online prediction. SLAQ uses also online prediction to avoid offline overhead, and leverages the iterative nature of ML training jobs to improve the accuracy of prediction.

Deadline-based scheduling. Many systems [46, 79, 114, 180] utilize scheduling to meet deadlines for batch processing jobs or to reduce lag for streaming analytics jobs. Jockey [84] uses a combination of offline prediction and dynamic resource allocation to ensure batch processing queries meet their latency SLAs while minimizing their impact on other jobs sharing the cluster. Instead of hard deadlines, some real-time systems [110, 184] use soft deadlines and penalize additional delay beyond the deadlines. However, these systems mainly consider the quality-runtime trade-offs for a single job, instead of optimizing across multiple approximate jobs.

3.8 Conclusion

In this chapter we present SLAQ, a quality-driven scheduling system designed for large-scale ML training jobs in shared clusters. SLAQ leverages the iterative nature of ML algorithms and obtains application-specific information to maximize the quality of models produced by a large class of ML training jobs. Our scheduler automatically infers the mod-

els' loss reduction rate from past iterations, and predicts future resource consumption and loss improvements online for subsequent allocation decisions. As a result, SLAQ improves the overall quality of executing ML jobs faster, particularly under resource contention.

Chapter 4

Riffle: Optimized Shuffle Service for Large-Scale Data Analytics

Large-scale data analytics systems are widely used in many companies holding and constantly generating big data. For example, the Spark deployment at Facebook processes 10s of PB newly-generated data every day, and a single job can process 100s of TB of data. Efficiently analyzing massive amounts of data requires underlying systems to be highly scalable and cost effective.

Data analytics frameworks such as Spark [191], Hadoop [2], and Dryad [105] commonly use a DAG of *stages* to represent data transformations and dependencies inside a *job*. A stage is further broken down to *tasks* which process different partitions of the data using one or more operations. Data transformations for grouping and joining data require all-to-all communication between *map* and *reduce* stages, called a *shuffle* operation. For example, a `reduceByKey` operation in Spark requires each task in the reduce stage to retrieve corresponding data blocks from all the map task outputs. Jobs that execute shuffle are prevalent—over 50% of Spark data analytics jobs executed daily at Facebook involve at least one shuffle operation.

The amount of data processed by analytics jobs is growing much faster than the memory available. At Facebook, data can be 10x larger than the total memory resource allocated to a job, and thus the shuffle intermediate data has to be kept on disks for scalability and fault tolerance purposes. The fast-growing data and complexity of analytics pose a fundamental performance tension in big-data systems.

Research work highly encourages running a large number of small tasks. Recent work [47, 115, 147–149] illustrates the benefit of slicing jobs into small tasks: small tasks improve the parallelism, reduce the straggler effect with speculative execution, and speed up end-to-end job completion. Solutions have also been presented to minimize task launch time [132] as well as scheduling overhead [150] for a large number of small tasks.

However, engineering experience often argues against running too many tasks. In fact, large jobs processing real-world workloads observe significant performance degradation because of excessive shuffle overhead [6, 7, 40]. While the tiny tasks execution plan works well with single-stage jobs, it introduces significant I/O overhead during shuffle operations in multi-stage jobs. Engineers often execute jobs with fewer bulky, slow tasks to mitigate shuffle overhead, paying the price of stragglers and inefficient large tasks that do not fit in memory.

We observe that the root cause of the slowdown is due to the fact that the number of shuffle I/O requests between map and reduce stages *grows quadratically* as the number of tasks grows, and the average size per request actually *shrinks linearly*. At Facebook, data is preserved on spinning disks for fault tolerance, so a large amount of small, random I/O requests (e.g., 10s or 100s of KB) during shuffle leads to a significant slowdown of job completion and resource inefficiency. Executing jobs with large numbers of tasks over splits the I/O requests, further aggravating the problem. Thus, neither approach for tuning the number of tasks provides efficient performance at large scales.

We present Riffle, an optimized shuffle service for big-data analytics frameworks that significantly improves I/O efficiency and scales to processing PB-level data. Riffle boosts shuffle performance and improves resource efficiency by converting large amounts of small, random shuffle I/O requests into much fewer large, sequential I/O requests. At its core, Riffle consists of a *centralized scheduler* that keeps track of intermediate shuffle files and dynamically coordinates merge operations, and a *shuffle merge service* which runs on each physical cluster node and efficiently merges the small files into larger ones with little resource overhead.

Challenges and solutions. In designing Riffle, we had to overcome several technical challenges.

First, Riffle has to be efficient in handling shuffle files without using much computation or storage resources. Riffle overlaps the merge operations with map tasks, and always accesses large chunks of data sequentially with minimal disk I/O overhead when performing merge operations. To reduce the additional delay caused by stragglers, Riffle allows users to set a *best-effort merge* threshold, so that reducers consume some late-arriving intermediate outputs in unmerged form, together with the majority of outputs in merged form.

Second, Riffle should be easy to configure to best fit different storage systems and hardware. While merging files generally reduces the number of I/O requests, making the block sizes too large leads to only marginal improvement in I/O requests but slowdown in merge operations. Riffle explores the inherent trade-off between maximizing the gain of large request sizes and minimizing the overhead of aggressive merges, and supports merge policies with different fan-ins and target block sizes, to get the best efficiency for disk I/Os and merge operations.

Third, Riffle must tolerate failures during merge and shuffle. Since failure is the norm at large scale, we must handle failures without affecting correctness or incurring additional slowdown in job execution. Riffle keeps track of intermediate files in both merged and

unmerged forms, and on failure falls back to files in unmerged format within the scope of failure.

Finally, Riffle should not create prohibitive overhead. The merge operations of Riffle come at the cost of reading and writing more shuffle data for the merged intermediate files. Riffle makes this trade-off a performance win, by issuing all merge requests as large, sequential requests, keeping the overhead significantly less than the savings. In terms of space, the intermediate files are soon garbage collected after job completion, so they occupy disk space only temporarily.

We implemented the Riffle shuffle service within the Apache Spark framework [5]. Riffle supports unmodified Spark applications and SparkSQL queries [51]. This paper presents the results of Riffle on a representative mix of Facebook's production jobs processing 100s of TB of data: Riffle reduces disk I/O requests by up to 10x and the end-to-end job completion time by up to 40%.

4.1 Background and Motivation

The past several years has seen a rapid increase in the amount of data that is being generated and analyzed every day. Distributed data analytics engines, like Spark [191], MapReduce [81], Dryad [105], are widely used for executing SQL queries and user-defined functions (UDFs) on large datasets, or preprocessing and postprocessing in machine learning jobs. The key challenge in analyzing massive amounts of data arises from the fact that the volume and complexity of data processing is growing much faster than hardware speed and capacity improvements. Riffle aims to solve the problem at large scale by significantly improving the efficiency of hardware resource usage.

This section motivates and provides background for Riffle. §4.1.1 briefly reviews the DAG computation model commonly used in big-data analytics frameworks. §4.1.2 discusses the memory constraints of data processing, and the quadratic relationship between

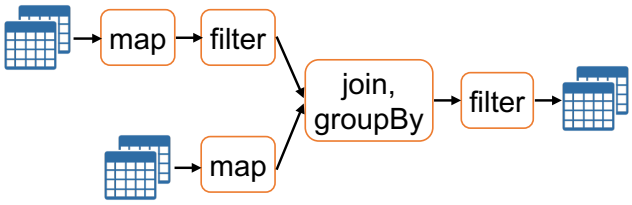
data volume and disk I/O during shuffle. §4.1.3 presents existing solutions to mitigate the problem, and explains why they fall short in fundamentally solving the problem at large scale.

4.1.1 Shuffle: All-to-All Communications

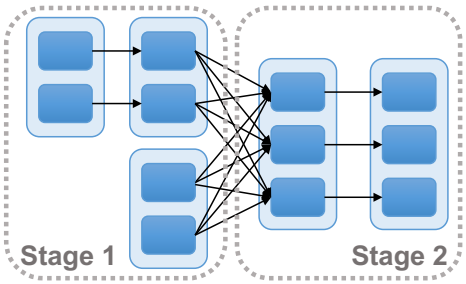
Data analytics frameworks typically use a DAG to represent the computation logic of a *job*, with stages as its vertices, and the dependencies between stages as its edges. A *stage* is further comprised of a set of tasks, each processing a partition of the datasets. A *task* typically includes a pipeline of one or more programmer specified operators that need to be applied to transform a data partition from input to output. Tasks in the first and last stages of a job are responsible to read in data from external sources (e.g., file systems, table storage, streams) and persist results, while tasks in the middle stages take the output generated by tasks in the previous stage as input, perform the transformation based on the specified operators, and then generate data for tasks in the next stage. Data dependencies thus can be classified in two types [191]: *narrow dependencies*, where the partition of data processed by a child task only depends on one parent task output, and *wide dependencies*, where each child task processes outputs from multiple or all parent tasks.

For example, Figure 4.1(a) is a logical view of a Spark job. It applies transformations (`map` and `filter`) on data from two separate tables, `join`s and aggregates the items over each key (a certain field of items) using `groupByKey`. After `filter`ing, it stores the output data in the result table. Figure 4.1(b) shows the Spark execution plan of this job. For narrow dependencies (`map` and `filter`), Spark pipelines the transformations on each partition and performs the operators in a single stage. Internally, Spark tries to keep the intermediate data of a single task in memory (unless the size of data cannot fit), so the pipelined operators (a `filter` operator following a `map` operator in Stage 1) can be performed efficiently.

Spark triggers an all-to-all data communication, called *shuffle*, for the wide dependency between Stages 1 (map) and 2 (reduce). Each map task reads from a data partition (e.g.,



(a) Logical operators.



(b) DAG execution plan.

Figure 4.1: DAG representation of a Spark job, which `joins` data processed from two tables and uses `groupByKey` to aggregate the key-value items, then `filters` the data to get the final results.

several rows of a large table), transforms the data into the intermediate format with the map task operators, sorts or aggregates the items by the partitioning function of the reduce stage (e.g., key ranges) to produce *blocks* of items, and saves the blocks to on-disk intermediate files. The map task also writes a separate *index file* which shows the offsets of blocks corresponding to each reduce task. To organize reduce stage data with `groupByKey`, each reduce task brings together the designated data blocks and performs reduce task operators. By looking up the offsets in index files, each reduce task issues fetch requests to the target blocks from all the map output files. Thus, data that was originally partitioned according to table rows are processed and shuffled to data partitioned according to reduce key ranges.

The large amount of intermediate files, written by the map tasks and read by the subsequent reduce tasks, are persisted on disks in both Spark and MapReduce for fault tolerance purposes. For large jobs, 10s to 100s of TB, or even PB of data are generated during each shuffle. Between stages with wide dependencies, each reduce task requires reading data blocks from all the map task outputs. If the intermediate shuffle files were not persisted, even a single reduce task failure could lead to recomputing the entire map stage. In fact,

failure of tasks or even cluster nodes is the norm at large scale deployment of big-data frameworks [102, 118, 182], so it is crucial to persist shuffle data for strong fault tolerance.

As a result, shuffle is an extremely resource intensive operation. Each block of data transferred from a map task to a reduce task needs to go though data serialization, disk and network I/O, and data deserialization. Yet shuffle is heavily used in various types of jobs—those requiring data to be repartitioned, grouped or reduced by key, or joined all involve shuffle operations. At Facebook, we observe that over 50% of our daily batch analytics jobs have at least one shuffle operations. A key approach to better completion time and resource efficiency of these jobs is improving the performance of shuffle operations.

4.1.2 Efficient Storage of Intermediate Data

Even though there is a trend towards keeping data in memory wherever possible to improve resource efficiency [4, 75, 83, 127], in real-world settings the amount of data is growing much faster than the available memory, which makes it infeasible to keep the data entirely in memory. For example, a job at Facebook processes data that is over 10x larger than the allocated resources. Instead intermediate data must be pushed to permanent storage for scalability and fault tolerance.

At Facebook, the current generation of warehouse clusters use HDDs for permanent storage. For large amount of data, this is significantly more cost effective than SSDs given current hardware [92, 117]. With spinning HDDs, the number of IOPS (I/O Operations Per Second) available is a limiting factor for the system throughput. While HDDs continue to grow in capacity, the available IOPS will not increase accordingly due to the physical limits of mechanical spin time [188]. Thus, we must be careful to use IOPS wisely for intermediate data, both for disk spills and shuffles.

Disk spill I/O. When the size of the data partition assigned to a task exceeds the memory limit, the task has to *spill* intermediate data to permanent storage. Disk spills can incur a

significant amount of additional overhead because of the increasing disk I/O and garbage collection.

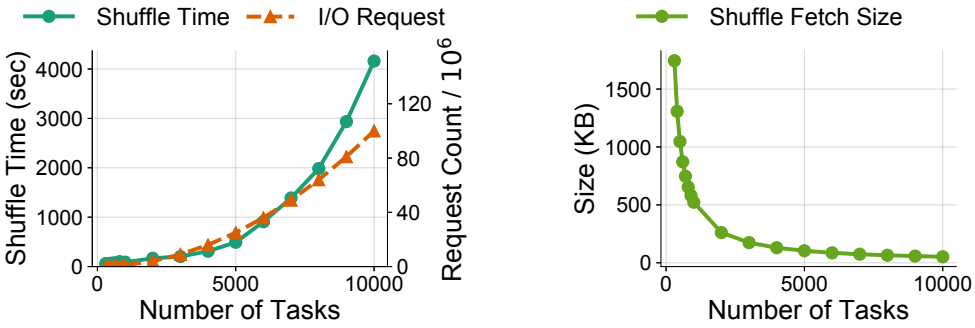
For example, assume that a map task processes a partition of 4GB input data, and runs with 8GB memory.[1] Data have to be decompressed and deserialized from disks to get the in-memory objects. This process effectively enlarges the original data, in practice, by about 4x. Thus, reading and processing 2GB input data already consumes the entire memory. The map task has to perform the operations and sort the result items by keys according to the reduce partition function. To do so, it (1) reads in the first 2GB data, performs the computation, and spills a temporary output file on disk; (2) similarly, reads, processes, and spills the second 2GB data; (3) merges the two temporary files into one using external merge sort. The overhead of repeated disk I/O and serialization significantly slows down the task execution and harms resource efficiency.

Shuffle I/O. To avoid disk spills, the task input size (S) should be appropriate to fit in memory, and thus is determined by the underlying hardware. As the size of job data increases, the number of map (M) and reduce (R) tasks has to grow proportionally. Because each reduce task needs to fetch from all map tasks, the number of shuffle I/O requests $M \cdot R$ increases *quadratically*, and the average block size $\frac{S}{R}$ for each fetch *decreases* linearly.

Figure 4.2 shows the job completion time when we keep the task input size fixed at 512MB (incurring no disk spills), and increase the number of tasks in both stages from 300 to $10,000$. We see that the shuffle time grows quadratically from 100 to over $4,000$ seconds. This is because the number of shuffle fetch requests increases rapidly (30K to 100M), as the average size of each request shrinks (1.7MB to 50KB).

Since disks are especially inefficient in handling large amounts of small, random I/O requests, jobs suffer a severe slowdown at large scale. Our goal is to improve the efficiency

[1] In practice, only a portion of memory can be used to cache data and the remaining is reserved by the runtime and program. The example ignores this discussion for simplicity.



(a) Shuffle time and I/O requests.

(b) Average I/O request size.

Figure 4.2: When the number of tasks in each stage grows, the shuffle time and the number of I/O requests increase quadratically, and the average shuffle fetch size in each request decreases.

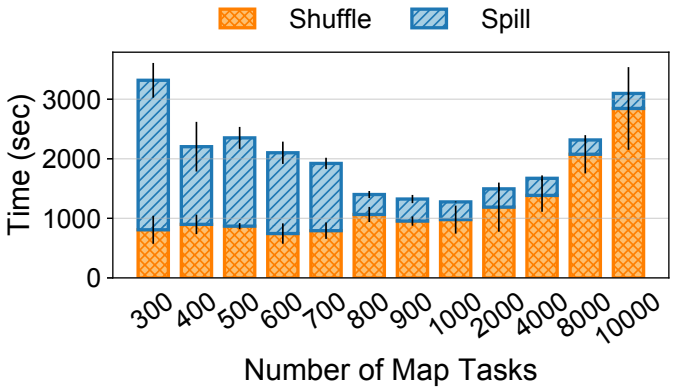


Figure 4.3: Shuffle-spill trade-off when varying number of map tasks (with fixed number of reduce tasks). Bulky tasks (left) incur more spill overhead, while tiny tasks (right) incur significant shuffle overhead.

by reducing the IOPS requirement of the underlying storage systems for large-scale data analytics.

4.1.3 Current Practices and Existing Solutions

Several solutions have been previously proposed to mitigate the problem of large amounts of small, random I/O requests during shuffle. We discuss the limitations of these solutions, and explain why they fall short in fundamentally solving the problem at large scale.

Reducing the number of tasks per stage. By tuning the number of tasks in job execution plan, engineers look for the optimal performance trading off between shuffle and spill I/O efficiency [7]. Since the number of I/O requests is determined by the corresponding map and reduce tasks, using fewer tasks reduces the total number of shuffle fetches, and thus improves the shuffle performance. However, this approach inevitably enlarges the average size of input data and creates very bulky, slow tasks with disk spilling overhead.

For example, Figure 4.3 shows how the shuffle and spill runtime changes when varying the number of map tasks in a job processing 3TB data. Towards the left, smaller number of tasks implies larger task partition sizes, making the shuffle operations more efficient. At the same time, larger tasks also mean each task needs to spill more data, slowing down the task completion time. In this case, at around 1,000 tasks the job reaches its optimal value in terms of the total runtime of shuffle and spill.

However, tuning the number of tasks is untenable to apply across the thousands of jobs at Facebook. Each job has different characteristics (e.g., distribution and skew of data), so it is not possible to find the optimal point without tedious experimentation. In addition, data characteristics change over time, depending on outside factors such as Facebook user behavior. Jobs are typically configured in favor of having more tasks, which allows room for data growth.

More importantly, the effects of having a small number of bulky tasks can be very detrimental for job execution in production: such tasks run very slowly due to additional I/O and garbage collection overhead [148]. In practice we see that task number tuning could assign GBs of data to a single task, causing the tasks to run over 60 minutes. Bulky tasks amplify the straggler problem, in that jobs get significantly delayed if a few tasks become stragglers or retry after failure, and speculative execution can only provide limited help in these cases [47, 149].

Aggregation servers for reducers. Another solution is to use separate aggregation processes in front of each reducer to collect the fragmented shuffle blocks and batch the disk I/O for shuffle data. The in-memory buffering in the aggregators ensures sequential disk access when writing shuffle data, which can later be read by reduce tasks all at once. However, directly applying this approach to process 100s of TBs or PBs of data is still infeasible. One aggregator instance per reduce task could consume a large amount of computation (for task bookkeeping) and memory (for disk I/O buffering) resources for large jobs, so the solutions can only be applied at relatively small scale [159]. In addition, because each reduce task collects data from all the map tasks, even failure of a single aggregation process leads to data corruption and requires the entire map stage to be recomputed. As jobs further scale in number of processes and runtime, the frequency of aggregation process failures (due to machine or network failures, etc.) increases. The high cost of failure recovery makes the solution inadequate for deployment at large scale.

To improve Hadoop shuffle performance, Sailfish [157] leverages a new file system design to support multiple insertion points to store aggregated intermediate files. Besides the fact that it requires modification to file systems, the solution also impairs the fault tolerance: to recover a single corrupted aggregation file, a large number of map tasks need to be re-executed. Compromising fault tolerance leads to frequent re-computation and thus harms system performance at Facebook scale.

Instead of trading fault tolerance for I/O efficiency, our goals of designing an optimized shuffle service include highly efficient shuffle I/O performance, little resource overhead to the clusters, and no additional failures caused by the shuffle optimization. Riffle provides its service as a long-running process on each physical node, and requires much less memory space and almost no computation overhead compared to existing solutions. Riffle operates on persisted disk files and saves results as separate files, so the service failures will not lead to any recomputation of stages or tasks. In the rest of the paper, we will show how Riffle's design and implementation meet these design goals in detail.

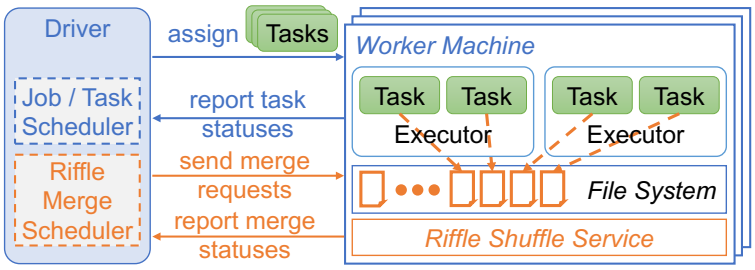


Figure 4.4: Riffle runs a shuffle merge scheduler as part of the analytics framework driver, and a merger instance per physical node. Since a physical node is typically sliced into a few executors, each running multiple tasks, it's common to have hundreds of tasks per job executed on each node.

4.2 System Overview

Riffle is designed to work with multi-stage jobs running on distributed data analytics frameworks that involve all-to-all data shuffle between different stages. We describe how Riffle works with cluster managers and data analytics frameworks, as shown in Figure 4.4.

Shuffle merge scheduler. Tasks in data analytics frameworks are assigned by a global driver program. As explained in §4.1, the driver converts a data processing job to a DAG of data transformations, with several stages separated by shuffle operations. Tasks from the same stage can be executed in parallel on the executors, while tasks in the following stage typically need to be executed after the shuffle. The intermediate shuffle files are persisted on local or distributed file systems (e.g., HDFS [164], GFS [86], and Warm Storage [16]).

Riffle includes a shuffle merge scheduler on the driver side, which keeps track of task execution progress and schedules merge operations based on configurable strategies and policies. In practice, it is common to have hundreds of tasks assigned per physical node in processing large-scale jobs. The Riffle scheduler collects the state and block sizes of intermediate files generated by all tasks, and issues merge requests when the shuffle files meet the merge criteria (§4.3.1).

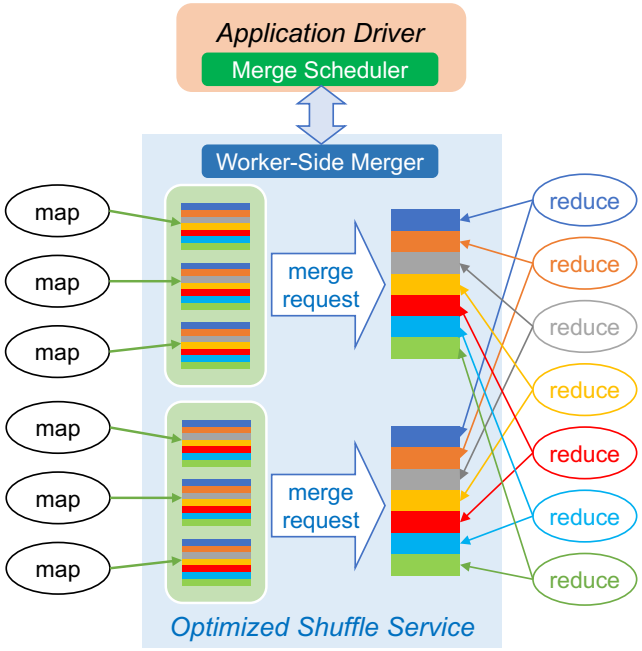


Figure 4.5: Merging intermediate files with Riffle.

Shuffle service with merging. Data analytics frameworks provide an external shuffle service [25, 35] to manage the shuffle output files. A long-running shuffle service instance is deployed on each worker node in order to serve the shuffle files uninterruptedly, even if executors are killed or reallocated to other jobs running concurrently on the cluster with dynamic resource allocation policies [87, 100]. Riffle runs a merger instance as part of the shuffle service on each physical node, which performs merge operations on shuffle output files.

The shuffle merge scheduler directly communicates with all the registered merger instances where some of the job tasks are executed, to send out merge requests and collect results from the mergers. Figure 4.5 illustrates the shuffle service side merger combining multiple intermediate shuffle files into larger files. Each mapper outputs data such that items are partitioned into the reducer it belongs to (indicated here by color). Without Riffle, each reducer would read partitions from all map outputs, which can be on the order of tens of thousands per reducer. Riffle merges the shuffle files block by block to preserve the reducer partitioning. After the merge operations, a reducer only needs to fetch a significantly smaller

number of large blocks from the merged intermediate files instead. Note that these merge operations are performed on compressed, serialized data files. This process significantly improves the shuffle I/O efficiency without incurring much resource overhead.

4.3 Design

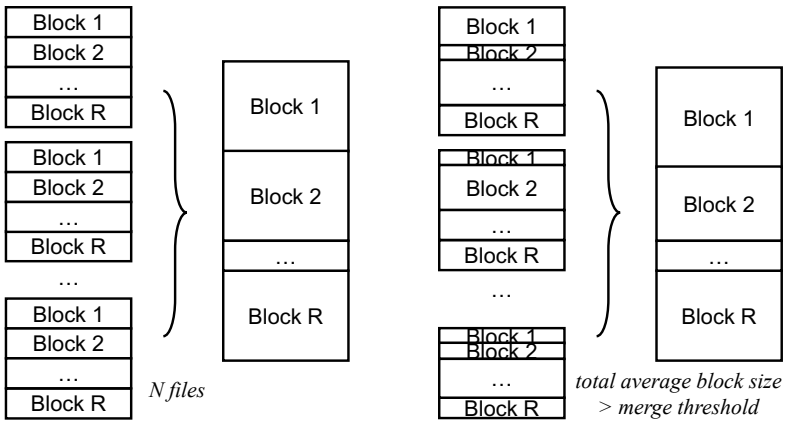
This section describes the mechanisms by which Riffle addresses its key challenges. We explain the merge strategies and policies in the driver side scheduler, and the execution of merge operations in the worker side merger in §4.3.1. We discuss how Riffle minimizes the merge overhead with best-effort merging (§4.3.2), handles merge failures (§4.3.3), and balances merge requests using power of choices in disaggregated architecture (§4.3.4). We analyze Riffle's performance benefit in §4.3.5.

4.3.1 Merging Shuffle Intermediate Files

Riffle is designed to work with existing data analytics frameworks by introducing shuffle merge operations in the shuffle service instances coordinated by the driver. Specifically, Riffle builds additional communication channels between the scheduler and mergers, allowing the driver to issue requests and coordinate mergers.

The merge scheduler starts merge operations immediately as map outputs become available, according to merge policies (§4.3.1). This causes most merges to overlap with the ongoing map stage, *hiding* their merge time if they finish before the map stage. When the map stage finishes, outstanding merge requests can incur additional delay, which makes policy configuration and merger efficiency (§4.3.1) important.

After the map tasks and merge operations finish, the driver launches reduce tasks in the subsequent stage, and broadcasts the metadata (location, executor id, task id, etc.) of all the map outputs to the executors hosting reduce tasks. With Riffle, the driver sends out



(a) Greedy N-way Merge. (b) Fixed Block Size Merge.

Figure 4.6: Riffle merge policies.

metatdata of the merged files instead of the original map output files, so the reduce tasks can fetch corresponding blocks from the merged files with more efficient reads.

Merge Scheduling Policies

Merge with fixed number of files. Users can configure Riffle to merge a fixed number of files. For N-way merge, the scheduler sends a merge request to the merger whenever there are N map output files available on that node (Figure 4.6(a)). The merger, upon receiving this request, performs the merge by reading existing shuffle files, grouping blocks based on reduce partitions, and generating a new pair of shuffle output file and index file as the merge result.

Merge with fixed block size. In real-world settings, we observe a large variance in block sizes of the shuffle output files (Figure 4.6(b)). Some shuffle blocks themselves are large enough, leading to few fragmented reads; some are very tiny and we need to merge tens or hundreds of them to make shuffle reads efficient. Riffle also supports fixed block size merge. In this case, the driver sends out a merge request when the accumulated average shuffle block size across all partitions exceeds a configurable threshold. The Riffle scheduler

avoids merging files that already have large blocks, and merges more files with tiny blocks for better I/O efficiency.

Configuring the merge policy. While merging files generally leads to more efficient shuffle, merging too aggressively can exacerbate the merge operation delay. Merge request processing is limited by the disk writing speed. For example, Riffle mergers achieve nearly the sequential speed at about 100MB/s when writing the merged files in our current deployment. The larger the merged output file is, the longer the merge operation will take. Riffle's file and block size based policies provide flexibility to trade off between shuffle and merge efficiency on a per-job basis.

In addition, these policies allow Riffle to be applied to file systems with different I/O characteristics. For example, if a file system provides 2MB unit I/O size, larger requests will be split into multiple 2MB chunk reads. Merging aggressively to get gigantic block files only provides marginal benefits for shuffle reads. In this case, Riffle's merge policy can be configured to a lesser number of files or smaller block size.

Efficient Worker-Side Merger

Upon receiving a merge request, the worker-side merger performs the merge operation and generates new shuffle files, as shown in Algorithm 2. A merge request includes a list of completed map task IDs. The merger locates the shuffle files previously generated by those tasks, and their accompanying index files which contain offsets of file blocks corresponding to individual reduce tasks. For each shuffle file, the merger allocates a buffer for asynchronously reads and caches its index file (normally no larger than a few tens of KB) in memory. The merger also allocates a separate buffer to asynchronously write the merged output file. During merge, it goes through each reduce partition, asynchronously copies over the corresponding blocks from all specified files into the merged file, and records the offsets in the merged index.

Algorithm 2 Merging Intermediate Shuffle Files

- *files*: shuffle files to be merged in request
- *index_files*: accompanying index files, which has offsets of shuffle file blocks corresponding to each reduce task
- *out_file*: merged shuffle file
- *out_index*: index file for the merged shuffle file
- *offset*: integer tracking offset of merged file

```
1:  function MERGESHUFFLEFILES(in_ids, out_id)
2:      for all id in in_ids do
3:          files[id] = OPENWITHASYNCREADBUFFER(id)
4:          index_files[id] = CACHEDINDEXFILE(id)
5:      out_file = OPENWITHASYNCWRITEBUFFER(out_id)
6:      out_index = NEWINDEXFILE(out_id)
7:      offset = 0
8:      for p = 1.. number of reduce partitions do
9:          for all id in in_ids do
10:             start = index_files[id].GETOFFSET(p)
11:             end = index_file[id].GETOFFSET(p + 1)
12:             length = end − start
13:             BUFFEREDCOPY(out_file, offset, files[id], start, length)
14:             offset = offset + length
15:             APPENDINDEX(out_index, offset)
16:     FLUSHBUFFERANDCLOSE(out_file)
17:     PERSISTINDEXFILE(out_index)
18:     return out_file, out_index
```

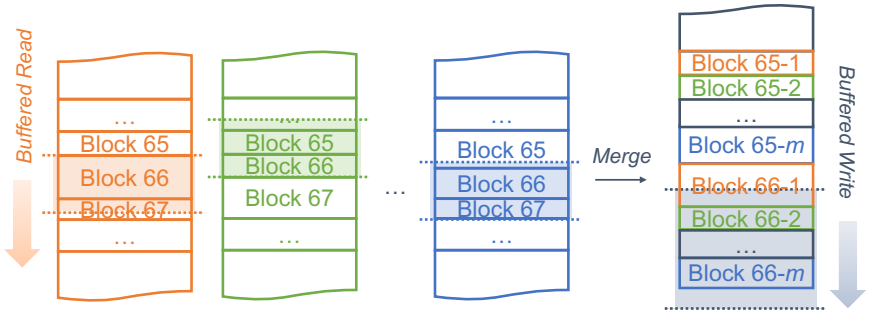


Figure 4.7: Riffle mergers trigger only sequential disk I/O for efficiency. The shadow sections of the input and output files are asynchronously buffered in memory to ensure sequential I/O behavior.

Riffle ensures the merge operation is efficient and lightweight on the worker side. First, Riffle merges compressed, serialized data files in their raw format on disks, incurring minimal computation overhead during merge. Second, the mergers prefetch data from original shuffle files, aggregate the blocks belonging to same reducers, and asynchronously write blocks into the result file. Thus, they always read and write large chunks of data sequentially, leading to minimal disk I/O overhead when performing merge operations.

Memory Management. The major resource overhead on the workers comes from the in-memory buffers for reading the original shuffle output files and writing the merged file, as shown in Figure 4.7. Buffering files ensures large, sequential disk I/O requests, at the cost of more memory consumption when the number of files and the number of concurrent merge operations grow.

For example, assume that we keep a 4MB read buffer and a 20MB write buffer. To merge 20 shuffle files, the merger has to buffer 80MB data for all input files, and 20MB for the output file, ending up consuming 100MB memory. Using a dedicated buffer for each file parallelizes the reads and writes and accelerates the merge speed. However, since a merger is responsible to handle hundreds of map output files per job generated by tens of executors on the worker node, the memory consumption can be significant when handling a large number of concurrent merge requests.

Riffle deploys mergers with a fixed memory allocation on each physical node. Upon receiving a new merge request, the merger estimates the memory consumption of processing the request based on the fan-in (i.e., number of files) and average block sizes, and only starts the operation if there is enough memory. When exceeding the memory limit, new incoming merge requests will be queued up and waiting for the memory to become available. We find that allocating 6–8GB of memory to a merger is sufficient to process 10–20 concurrent merge requests in most use cases.[2] With this configuration, Riffle mergers can achieve nearly sequential disk I/O speed when writing merged files. Given that each physical node typically has 256GB or even larger memory in modern datacenters, and tens of GB of memory per machine is reserved for OS and framework daemons, we consider the memory overhead of Riffle acceptable.

[2] The memory allocation of the merger determines the number of concurrent requests it can handle. In general, increasing the memory space leads to higher merge throughput, until a certain point where the effective disk output rate becomes a limiting factor.

4.3.2 Best-Effort Merge

When processing large-scale jobs with Riffle, there are usually some merger processes still working on performing merge operations while most of the other mergers have already completed the assigned requests. These *merge stragglers* exist mainly for two reasons. First, there are always shuffle files that are generated by the final few map tasks, and the late merge operations need to wait for these tasks to complete before starting to merge. Second, the mergers on the worker nodes could also crash and get restarted, which slows down the pending merge requests on that node. We find that when deployed at large scale, Riffle merge stragglers can sometimes significantly increase the end-to-end job completion time.

To alleviate the delay penalty caused by stragglers, we introduce *best-effort merge*, which allows the driver to mark the map stage as finished and start to launch reduce tasks when *most* merge operations are done on workers. Riffle allows users to configure a percentage threshold, and when the completed merge operations exceed this threshold, the driver does not wait for additional merge requests to return. The job execution directly proceeds to the reduce stage, and all pending merge operations are cancelled by the driver to save resources.

When using best-effort merge, the Riffle driver sends to reducers the metadata for merged shuffle files for successful merge operations, and the metadata of original unmerged files for cancelled merge operations. By eliminating merge stragglers, best-effort merge improves the end-to-end job completion time as well as the overall resource efficiency despite a small portion of shuffle fetches being done on less efficient unmerged files. We demonstrate this improvement in §4.5.2.

4.3.3 Handling Failures

Since failure is the norm at scale, Riffle must guarantee the correctness of computation results, and should not slow down the recovery process when failures happen. This requires Riffle to efficiently handle both merge operation failures and loss of shuffle files. To handle

these cases well, Riffle keeps both the original, unmerged files as well as the merged files on disks.

A merge operation can fail if the merge service process crashes, or merging takes too long and the request times out. When that happens, Riffle is designed to fall back to original unmerged files in similar manner to best-effort merge. This leads to a slight performance degradation during shuffle, while avoiding delaying the map stage. Correctness is guaranteed in the same way as best-effort merge, by the Riffle driver sending a mixture of metadata for merged and unmerged files to reduce tasks.

Spark and Hadoop deal with shuffle data loss or corruption by recomputing only the map tasks that generated the lost files. Riffle follows this strategy if unmerged files are lost, but can recover faster if only merged files are lost. For lost merged files, the original shuffle file is used as a fallback, avoiding any recomputations in the map stage while slightly degrading shuffle by fetching more files. Note that this is different from previous solutions using aggregators to collect data on the reducer side. Sailfish [157] modifies the underlying file system with a new file format that supports multiple insertion points for reduce block aggregation. However, a data loss which involves a single chunk of the aggregated file requires re-execution of all map tasks which appended to that chunk. Thus, data losses can lead to heavy recomputation for the tasks in the map stage, and it falls short to meet our key requirement of efficient failure handling.

4.3.4 Load Balancing on Disaggregated Architecture

Recent development in datacenter resource disaggregation [15, 85, 130] replaces individual servers with a rack of hardware as the basic building block of computing. The new-generation disaggregated architecture provides efficiency through gains in flexibility, latency, and availability. At Facebook, disaggregated clusters are widely used: the compute nodes (with powerful CPUs and memory) and storage nodes (with weaker CPUs and large disk space) on separate racks. The distributed file system abstracts away the physical file

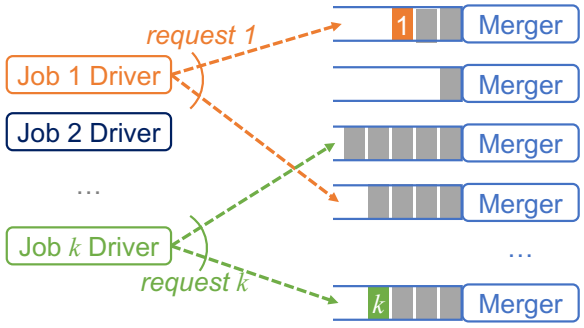


Figure 4.8: Multiple Riffle jobs on a disaggregated architecture balances the merge requests leveraging the power of two choices.

locations, and leverages fast network connections to achieve high I/O performance across all storage nodes. While deploying a data analytics framework such as Spark on the disaggregated clusters, all workers experience nearly homogeneous rates reading and writing files regardless of their physical locations in the storage nodes.

Riffle on disaggregated clusters runs one merger process on each compute node. In the context of resource disaggregation, merge operations are no longer limited to work with "local" shuffle files generated from the same physical nodes. In fact, the driver can send a request to an arbitrary merger to merge a number of available shuffle output files generated by multiple executors on different physical nodes. For example, when the fixed block size policy is used, the driver will pick a merger and send out a merge request whenever the accumulated average block sizes of shuffle files generated by all workers exceed the minimum merging block size.

Because of the merger memory limits, merge requests can queue up when the cluster experiences high workload (as described in §4.3.1). Note that the mergers, located on the physical nodes, are shared across all concurrent jobs running on the cluster. The Riffle enabled drivers need to consider the workload of mergers when sending out their requests, so that the merge operations are balanced among the mergers.

In order to efficiently balance the dynamic merge workload in a distributed manner, Riffle leverages "power of two choices" [141]. As shown in Figure 4.8, each driver only needs to query the pending merge workload of two (or a few) randomly picked mergers and

choose the one with the shortest queue length. Theoretical analysis and experiments [135, 150] show that the approach can efficiently balance the distributed dynamic requests while incurring little probing overhead.

4.3.5 Discussion

Analysis of I/O operation savings. Assume a two-stage job has M map tasks and R reduce tasks. The total amount of data it processes is T. To simplify the discussion, we assume the partition processed by each task can fit in memory (i.e., no disk spills). With unmodified shuffle, the number of total shuffle I/O requests is $M \cdot R$.

Using N-way complete merge, $\frac{M}{N}$ merged files are generated by the mergers. During shuffle, each reducer only sends $\frac{M}{N}$ read requests. Assuming data is evenly partitioned, the total shuffle I/O requests during is now $\frac{M}{N} \times R$.

Merge operations also trigger additional I/O. Specifically, a complete merge of all intermediate files requires an additional read and write of T data. Since Riffle mergers only incur sequential disk I/O, the total number of I/O requests is $2 \cdot \frac{T}{s}$, where s is the buffer size in the Riffle mergers. Putting them together, the total number of I/O requests is

$$\frac{M}{N} \times R + 2 \cdot \frac{T}{s}$$

For example, assume a job processing 100GB data uses 1,000 map tasks and 1,000 reduce tasks. It triggers 1,000,000 I/O requests during shuffle. If the Riffle merger uses 10MB I/O buffers, then with 40-way merge, the total number of I/O requests becomes $\frac{1000}{40} \times 1000 + 2 \times \frac{100GB}{10MB}$ =45,000, reduced by 22x.

This calculation does not consider the effect of disk spills. In fact, Riffle's efficient merge alleviates the quadratic increase of shuffle I/O. Thus users can run much smaller tasks instead of bulky tasks, which further reduces disk IOPS requirement due to less spills.

Note that the amount of additional I/O incurred by Riffle is similar to that required in Sailfish [157]. More specifically, the *chunkserver*s and *chunksorter*s in Sailfish also need to make a complete pass reading and writing shuffle data to reorganize the key-values and generate new index files. Both systems move this process off the critical path to unblock the execution of map and reduce tasks. Riffle's configurable merge policy and best-effort merge mechanism further minimize the merge overhead. In contrast, ThemisMR [158] provides exactly twice I/O property. Compared with Riffle and Sailfish, it completely avoids materializing intermediate files to disks, at the cost of impaired fault tolerance. Thus, the solution only applies to relatively small scale deployment.

Deployment on different clusters. Riffle works best when there are multiple executors processing tasks on each physical machine. As computing nodes getting larger and more powerful, it is desirable to slice them into smaller executors for efficient resource multiplexing (i.e., shared by multiple concurrent jobs) and failure isolation. In addition, Riffle fits well with recent research and industry trends in resource disaggregation, where merge operations are no longer limited to "local" files (§4.3.4). Large jobs running on small machines can still benefit from Riffle: in this case, tasks in map stage come in *waves*, ending up with many shuffle files on each physical node to merge.

4.4 Implementation

We implemented Riffle with about 4,000 lines of Scala code added to Apache Spark 2.0. Riffle's modification is completely transparent to the high-level programming APIs, so it supports running unmodified Spark applications. We implemented Riffle to work on both traditional clusters with collocated computation and storage, and the new-generation disaggregated clusters. Riffle as well as its policies and configurations can be easily changed on a per-job basis. It is currently deployed and running various Spark batch analytics jobs at Facebook.

Garbage collection. Storage space, compared to other resources, is much cheaper in the system. As described in §4.3.3, Riffle keeps both unmerged and merged shuffle output files on disks for better fault tolerance. Both types of shuffle output files share the lifetime of the running Spark job, and are cleaned up by the resource manager when the job ends.

Correctness with compressed and sorted data. Compression is commonly used to reduce I/O overhead when storing files on disks. The data typically needs to go through compression codecs when transforming between its on-disk format and in-memory representation. Riffle concatenates file blocks directly in their compressed, on-disk format to avoid compression encoding and decoding overhead. This is possible because the data analytics frameworks typically use concatenation friendly compression algorithms. For example, LZ4 [24] and Snappy [34] are commonly used in Spark and Hadoop for intermediate and result files.

Merging the raw block files breaks the relative ordering of the key-value items in the blocks of merged shuffle files. If a reduce task does require the data to be sorted, it cannot assume the data on the mapper side is pre-sorted. Sorting in Spark (default) and Hadoop (configurable) on reduce side uses the TimSort algorithm [37], which takes advantage of the ordering of local sub-blocks (i.e., segments of the concatenated blocks in merged shuffle files) and efficiently sorts them. The algorithm has the same computational complexity as Merge Sort and in practice leads to very good performance [8]. The sorting mechanism ensures that reducer tasks will get the correctly ordered data even with the Riffle merge operations. In addition, since merge will not affect the internal ordering of data in sub-blocks (i.e., sorted regions in map outputs), the sorting time using TimSort with Riffle will be the same as the no merge case.

	Data	Map	Reduce	Block
1	167.6 GB	915	200	983 K
2	1.15 TB	7,040	1,438	120 K
3	2.7 TB	8,064	2,500	147 K
4	267 TB	36,145	20,011	360 K

Table 4.1: Datasets for 4 production jobs used for Riffle evaluation. Each row shows the total size of shuffle data in a job, the number of tasks in its map and reduce stages, and the average size of shuffle blocks.

4.5 Evaluation

In this section, we present evaluation results on Riffle. We demonstrate that Riffle significantly improves the I/O efficiency by increasing the request sizes and reduces the IOPS requirement on the disks, and scales to process 100s of TB of data and reduces the end-to-end job completion time and total resource usage.

4.5.1 Methodology

Testbed. We test Riffle with Spark on a disaggregated cluster (see §4.3.4). The computation *blade* of the cluster consists of 100 physical nodes, each with 56 CPU cores, 256GB RAM (with 200GB allocated to Spark executors), and connected with 10Gbps Ethernet links. Each physical node is further divided into 14 executors, each with 4 CPU cores and 14 GB memory. In total, the jobs run on 1,414 executors. 8GB memory on each physical node is reserved for in-memory buffering of the Riffle merger instance. The storage *blade* provides a distributed file system interface, with 100MB/s I/O speed for sequential access of a single file. Our current deployment of file system supports 512KB unit I/O operation. We also use emulated IOPS counters in the file system to show the performance benefit when the storage is tuned with larger optimal I/O sizes.

Workloads and datasets. We used four production jobs at Facebook with different sizes of shuffle data, representing small, medium and large scale data processing jobs, as shown

in Table 4.1. To isolate the I/O behavior of Riffle, in §4.5.2 we first show the experiment results on synthetic workload closely simulating Job 3: the synthetic job generates 3TB random shuffle data and uses 8,000 map tasks and 2,500 reduce tasks. With vanilla Spark, each shuffle output file, on average, has a $3\text{TB}/8000/2500 = 150\text{KB}$ block for each reduce task (approximating the 147KB block size in Job 3). Without complex processing logic, experiments with the synthetic job can demonstrate the I/O performance improvement with Riffle. We further show the end-to-end performance with the four production jobs in §4.5.3.

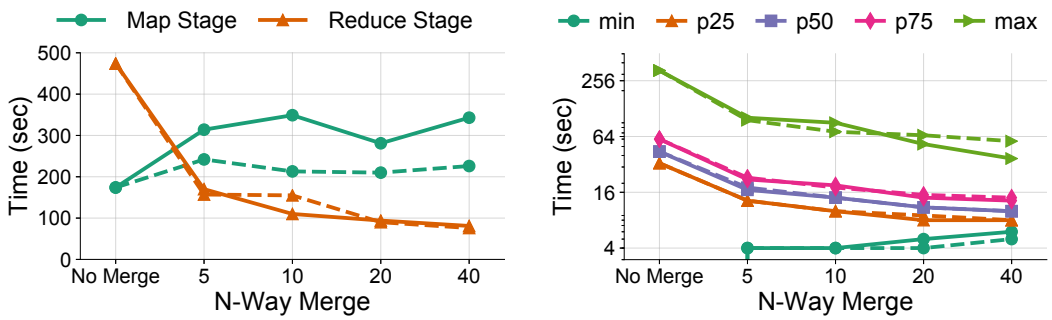
Metrics. Shuffle performance is directly reflected in the reduce task time, since each reduce task needs to first collect all the blocks of a certain partition from shuffle files, before it can start performing any operations. To show the performance improvement of Riffle, we focus on measuring (i) task, stage, and job completion time, (ii) reduction in the number of shuffle I/O requests, and (iii) the total resource usage in terms of reserved CPU time and estimated disk IOPS requirements.

Baseline. In the experiments with the synthetic workload, we compare the time and resource efficiency of Riffle with different merge policies. In the experiments with real-world workloads, we compare the performance improvement of Riffle against the engineering tuned execution plans (numbers of map and reduce tasks in Table 4.1) that have best shuffle-spill trade-offs with vanilla Spark.

4.5.2 Synthetic Workload

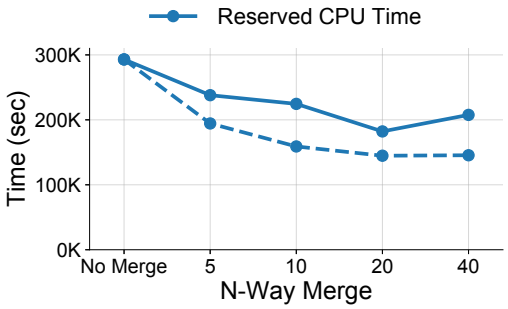
Stage and Task Completion Time

We compare the performance improvement of Riffle when doing 5-way, 10-way, 20-way and 40-way merge, respectively. The merged shuffle files will on average get 750KB, 1.5MB, 3MB and 6MB block sizes.



(a) Stage runtime.

(b) Reduce task runtime.



(c) Reserved CPU time.

Figure 4.9: Riffle performance improvement in runtime with synthetic workload. 4.9(a) and 4.9(b) show the wall clock time to complete stages and tasks, and 4.9(c) plots the total reserved CPU time representing the job resource efficiency. Map time includes time to execute both map tasks and Riffle merge operations. Reduce time includes time to perform both shuffle fetch and reduce tasks. No complex data processing is in the synthetic applications, so shuffle fetch dominates the reduce time. Dashed lines show the performance with best-effort merge.

Map and reduce stage execution time. Figure 4.9(a) shows the map and reduce stage completion time when running the job with vanilla Spark ("no merge") vs. Riffle with different merge policies (note the log scale on x-axis). As N grows, the merge operation generates larger block files, yet also takes longer time to finish. Since Riffle merge operations block the execution of reduce stage, map is only considered as completed when the merge is done. We see the map stage time increases gradually from 174 to 343 seconds. Despite the delay in map, we have a much larger reduction in the reduce stage time, which drops from 474 to 81 seconds. Overall, the job completion time (i.e., sum of the two stages) drops from 648 down to 424 seconds, 35% faster.

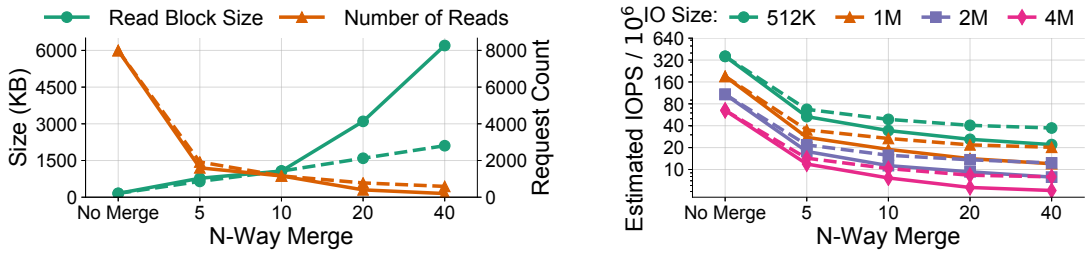
Improvement with best-effort merge. Riffle uses best-effort merge mechanism (§4.3.2) to further reduce the delay penalty of merge operations. In Figure 4.9(a), the dashed lines show the results of best-effort merge (threshold $= 95\%$). We can see the map stage overhead, compared to full merge, is much smaller (343 down to 226 seconds with 40-way merge), while the reduce stage time stays almost the same. Overall, the job completes 53% faster compared with vanilla Spark.

To better understand the reduce stage time improvement, we break down the stage time by plotting the distribution of all task completion time. We show the minimum, $25/50/75$ percentile, and maximum for different merge policies in Figure 4.9(b) (note the log scale of both axes). Similarly, results with best-effort merge are in dashed lines. The medium task time is reduced from 44 seconds (no merge) down to 10 seconds (40-way merge). The improvement comes from the fact that a reduce task only has to issue hundreds of large reads, as opposed to thousands of small reads, after the merge.

Improvement in Resource Efficiency

We measure the resource efficiency via metrics reported by the cluster resource manager. Figure 4.9(c) shows the total *reserved CPU time*. When merge is disabled, the entire job takes 293K reserved CPU seconds to finish; with over 20-way merge, the reserved CPU time is reduced to 207K seconds, or by 29%. In addition, when we enable best-effort merge, the saving in job completion time is also reflected in the resource efficiency—the total reserved CPU time is further decreased down to 145K seconds. That means we can finish the job with only 50% of the computation resource.

Note that the synthetic workload rules out the heavy data computation from the jobs, in order to isolate the I/O performance during shuffle. With production jobs, the overall resource efficiency also highly depends on the nature of the specific data processing logic. However, we expect to see the same resource efficiency gains when considering the shuffle operations alone.



(a) Shuffle I/O requests.

(b) IOPS requirement with different unit I/O sizes.

Figure 4.10: Riffle I/O performance during shuffle. The dashed lines show best-effort merge performance.

I/O Performance

Figure 4.10(a) demonstrates how the number and size of shuffle fetch requests change with different merge policies. The average read size (left y-axis) increases from 150KB (no merge) to up to 6.2MB (40-way merge), and the number of read requests (right y-axis) decreases from 8,000 down to 200. With best-effort merge, since shuffle files are partially merged, each reduce task still has to read 5% of data from the unmerged block files. With 40-way merge, we observe an average of 589 read requests per task, and the average read request size of 2.1MB. Riffle effectively reduces the number of fetch requests by 40x (10x) with complete (best-effort) merge.

To show the performance implication of the underlying file system, we look at the IOPS requirement for running the job with different policies. We measure the IOPS requirement with 512KB unit I/O size provided in our current deployment, and the estimated IOPS counters when the file system supports larger I/O sizes. Figure 4.10(b) shows how the shuffle IOPS changes (note the log scale of both axes) with different merge policies. We can see that Riffle reduces the job IOPS from 360M with no merge down to 22M (37M), or by 16x (9.7x), with complete (best-effort) 40-way merge. We see the 10x reduction carries over as we increase the file system I/O sizes to 1M, 2M or even larger.

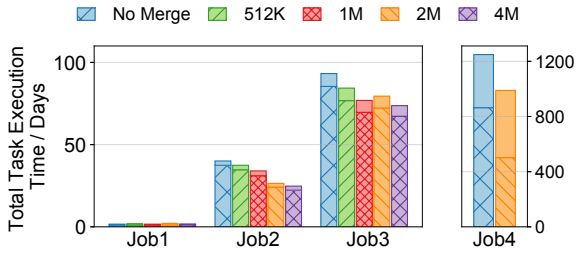
4.5.3 Production Workload

In this section, we demonstrate Riffle's improvement in processing 4 production jobs, representing small (Job 1), medium (Job 2 and Job 3), and large (Job 4) jobs at Facebook Compared with synthetic workload, the production jobs are different in several ways:

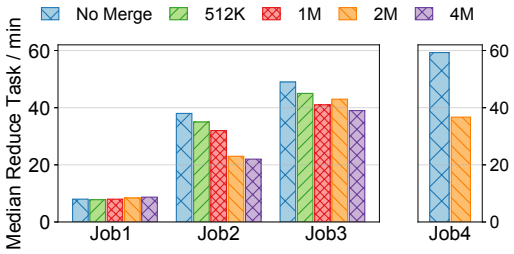
- They involve heavy computation in each task, instead of only I/O in the synthetic case;
- Jobs are deployed in real settings with limited memory resources that best fit the hardware configurations, and data will be spilled to disks if the memory space is insufficient;
- The block sizes of the intermediate shuffle files vary based on the user data distribution and the partitioning functions, and Riffle should merge based on block sizes instead of a fixed fan-in.

Improvement in I/O performance and end-to-end job completion time is crucial to production workload. For instance, Job 4 is processing a *key data set*, which is in the most upstream data pipeline for many other jobs under the same namespace. It processes hundreds of TB of data and consumes over 1,000 CPU days to finish. Accelerating this job will not only improve resource efficiency significantly, but also help improve the landing time of many subsequent jobs. We show the performance of Riffle with fixed block size merge, varying the block size threshold (512KB, 1MB, 2MB, and 4MB for first three jobs, and 2MB for the last job). All the experiments enable best-effort merge with a threshold of 95%.

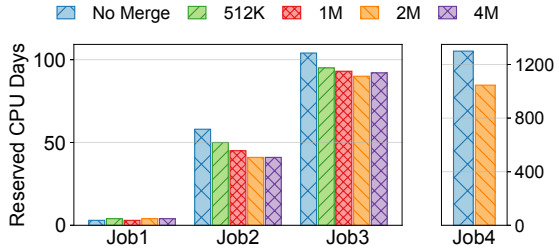
Stage and task completion time. Figure 4.11(a) shows that Riffle significantly helps decrease the reduce stage time by 20–40% for medium to large scale jobs, without affecting the map time much. Compared to the gain in synthetic workload, Riffle gets less relative time reduction because of the fixed computing cost in the tasks. Note that in the case of



(a) Map (top) and reduce (bottom) stage runtime.



(b) Median reduce task runtime.



(c) Reserved CPU time.

Figure 4.11: Riffle performance improvement with production workload.

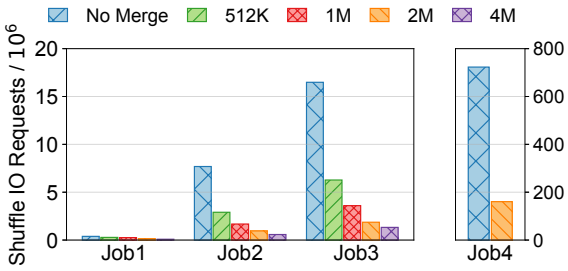


Figure 4.12: Number of shuffle I/O requests (million), including all additional I/O requests in Riffle mergers.

running small-scale jobs (like Job 1), Riffle does not help because of the delay penalty incurred by the additional merge requests. Figure 4.11(b) further explains that the saving of reduce stage time comes from shorter reduce task time. The reduce task can be shorten by up to 42% (39%) when running medium (large) scale jobs.

Resource efficiency. The big saving in job completion time leads to more efficient resource usage. Figure 4.11(c) measures the resource usage of running the jobs. We can see that Riffle in general saves 20–30% reserved CPU time for medium to large scale jobs.

Figure 4.12 compares the total I/O requests during shuffle. Riffle reduces the total shuffle I/O requests by 10x for Jobs 2 and 3, and by 5x for Job 4. For Jobs 2 and 3, Riffle effectively converts the average request size from the original 100–150KB (see Table 4.1) to 512KB or larger, and thus significantly reduces the number of read requests needed during shuffle operations. Similarly, for Job 4, Riffle increases the average 360KB reads to 2MB and thus reduces the number of I/O requests.

Riffle incurs additional I/O requests for merging shuffle files. The mergers use up to 64MB in-memory buffers to ensure that the merge operations only issue large, sequential I/O requests to disks. The overhead of merge I/O requests is almost negligible compared with the order of magnitude savings in shuffle I/O requests.

4.6 Related Work on Shuffle Optimization

Shuffle optimization in big-data analytics. ThemisMR [158] improves the performance of MapReduce jobs by ensuring the intermediate data (including shuffle and spill) are not repetitively accessed through disks. However, the solution does not avoid large amounts of small random I/O during shuffle. In addition, as the paper stated, ThemisMR eliminates the task-level fault tolerance, and thus only applies to relatively small scale deployment. TritonSort [160] minimizes disk seeks by carefully designing the layout of output files without using huge in-memory buffers. However, since it targets the specific sorting problem, the solution can hardly generalize to other data analytics jobs. Sailfish [157] leverages a new file system design to support multiple insertion points to aggregate intermediate files. However, it requires modifications to file systems, and a single corrupted aggregation file requires recomputation of a large number of map tasks.

Parameter tuning for data analytics frameworks. Previous work [58, 137, 190, 199] provide guidelines on how to best configure system parameters (such as number of tasks in each stage) with given cluster resources. Starfish [98] is a self-tuning system which

provides high performance without requiring users to understand the Hadoop parameters. However, the tuning process for a large number of jobs is expensive, and jobs have to be retuned when their characteristics such as the distribution and skew of input data change over time.

IOPS optimization. Sailfish [157] leverages a new file system design to support multiple insertion points to aggregate intermediate files. However, it requires modifications to file systems, and a single corrupted aggregation file requires recomputation of a large number of map tasks. Hadoop-A [186] accelerates Hadoop by overlapping map and reduce stages, and uses RDMA to speed up the data collection process. However, this solution relies on the reducer task to collect and buffer intermediate data in memory, which limits its scalability and fault tolerance. Recent development on hardware accelerates the handling of I/O requests and starts to get deployed in big-data analytics and storage systems [169, 185], but they do not targets the problem of small, random shuffle fetch for large-scale jobs.

The case for tiny tasks. Recent work [115, 148, 150] proposes tiny tasks which run faster and lead to better job completion time when investigating the performance of data analytics jobs. While solutions have been studied to minimize the task launch time [132] and overcome the scheduler overhead [150], tiny tasks hit the performance bottleneck of shuffle when used for large-scale jobs with multiple stages. Riffle merges intermediate files and significantly improves shuffle efficiency, so that the jobs can benefit from both fast task execution and efficient shuffle with small tasks.

Straggler mitigation. The original MapReduce paper [81] introduces the straggler problem. Previous work on data analytics leverages speculative execution [47, 49, 193] or approximate processing [43, 48, 178] to mitigate stragglers. Riffle avoids merge stragglers using best-effort merge, which allows shuffle files to be partially merged to avoid waiting for merge stragglers and accelerate job completion.

4.7 Conclusion

In this chapter we present Riffle, an optimized shuffle service for big-data analytics frameworks that significantly improves the I/O efficiency and scales to process large production jobs at Facebook. Riffle alleviates the problem of quadratically increasing I/O requests during shuffle by efficiently merging intermediate files with configurable policies. We describe our experience deploying Riffle at Facebook, and show that Riffle leads to an order of magnitude I/O request reduction and much better job completion time.

Chapter 5

Conclusion

5.1 Summary of Contributions

This dissertation contributes a number of techniques that achieve high efficiency and cost-effectiveness on cluster resource usage for advanced data analytics. These methods have been designed, implemented, and evaluated in production systems at large companies that analyze batch and stream data from real-world workloads and scales to process petabytes of data. Based on the idea of leveraging application-specific characteristics to design customized resource management systems, we manage to build highly-efficient, scalable, and fault-tolerant data frameworks for key scenarios including video analytics, distributed machine learning, and multi-stage batch processing. More specifically, this dissertation demonstrates the design and evaluation of the following systems.

- For *scheduling stream processing on video data*, we designed and built VideoStorm that allows users to submit queries with arbitrary vision processors. We efficiently generate resource-quality profiles for video queries without exhaustively exploring the combinatorial space of knob configurations. At its core, VideoStorm has an efficient scheduler for video queries that considers their resource-quality profile and lag tolerance, and trades off between them.

- For *scheduling distributed ML training jobs*, we built SLAQ, a fine-grained cluster scheduling system for running approximate ML training jobs on shared resources. Instead of targeting resource fairness, the system optimizes for overall training quality. SLAQ uses a normalization mechanism to unify quality measurement metrics even for different ML algorithms on different datasets. With the normalized quality metric, SLAQ's online prediction algorithm precisely estimate the quality and runtime for future iterations of the training process.
- For *improving I/O efficiency of task execution in large-scale batch processing*, we designed and implemented Riffle, which consists of a *centralized scheduler* that keeps track of intermediate shuffle files and dynamically coordinates merge operations, and a *shuffle merge service* which runs on each physical cluster node and efficiently merges the small files into larger ones with little resource overhead. Riffle resolves the contention between individual task efficiency and inter-stage I/O efficiency, and takes practical concerns like stragglers and fault tolerance into consideration.
- We implemented and deployed these systems, and evaluated the performance with real-world workloads we collected from collaborators such as operational cameras from the Bellevue Traffic Departments and production workloads from Facebook, as well as various online sources. Our experiments show significant improvement in system throughput and job completion time.

Our experience deploying these systems demonstrated the reliability and applicability of our approaches. VideoStorm is currently deployed and running in Bellevue, WA and Cambridge, U.K. for resource management on platforms processing live streams from thousands of operational traffic cameras. As the core resource manager, VideoStorm works with other layers of the analytics stack [23, 101, 109] together to make it easy and affordable to deploy real-time, low-cost, and accurate video analytics. Riffle is part of Facebook's Spark deployment, running on clusters with hundreds of physical machines, processing thousands of data analytics jobs daily on tens of petabytes of newly generated data per day.

Our experience running Riffle at Facebook show significant performance improvement on production workloads.

5.2 Open Issues and Future Work

Performance-driven scheduling for geo-distributed resources. Recent development in geo-distributed and edge computing efficiently handle large volumes of data originated from different sites. Previous work focus on aggregating and analyzing the geo-distributed data with limited bandwidth in the wide-area networks [155, 156, 183, 198]. By carefully placing the different tasks (operators) of a processing pipeline to different locations, the systems can best utilize the WAN bandwidth and thus reduce the data processing latency.

Multimedia data processing is a perfect match for geo-distributed data analytics. While VideoStorm processes video streams centralized or on-premises datacenters, the deployment requires sufficient WAN bandwidth through fiber drops or excessive cellular data. In fact, cameras are edge devices and can be equipped with chips for data preprocessing. One natural next step for VideoStorm is to support task and operator placement for video data considering the limitation of both the computation and network resources on edge devices and clusters. In addition to resource heterogeneity, multiple analytics queries can be kicked off concurrently. It presents an opportunity to jointly optimize query performance by carefully merging shared upstream operators in the processing pipelines, but also makes it harder for the planning of each query execution since the resource constraint becomes dynamic when shared with other queries. New scheduling algorithms should be designed to take the dynamic resource budget into consideration, and allocate resources to pipelined tasks to maximize the quality of the end results.

Leveraging approximate scheduling for automated hyperparameter tuning. Hyperparametre tuning is an important topic in machine learning. Many ML researchers have brought up with different approaches to explore the hyperpameter space and find the best

combination. Existing solutions include (1) grid search, which evenly samples trials across the configuration space; (2) random search, which randomly picks next points in the configuration space; and (3) Bayesion optimization [166], which chooses the next point with the highest probability that can lead to high model quality based on previous trials, with some assumptions of the quality distribution of the underlying configuration space. For each trial with a certain combination of hyperparameters, ML practitioners have to train the same model with a fixed number of iterations or until it converges.

We ask an orthogonal question from the system's perspective: how to improve the training efficiency on each individual trial? With SLAQ's prediction mechanism, we can automatically terminates those trainings that are not promising to beat the best result we've already got, and speed up the hyperparameter search process. To do so, an approximation method is required to precisely estimate the convergence point, so that the scheduler can compare the predicted quality with past trials and early stop non-promising trials. This could prevent a large amount of computation from being wasted on useless combinations, and accelerate those that generate models with quality close to the optimal. To make the system practical and beneficial to ML practitioners, the system needs to provide a set of clearly defined APIs and theoretical analysis to ensure a bounded quality of the result. For example, it would be helpful for the users to know that within a certain confidence range, the model from the approximate search system is at most 1% less accurate compared with the best possible model.

Automate the shuffle configuration for different deployment and workloads. Riffle and other related systems [157, 158] take different approaches to optimize the shuffle performance with vastly different assumptions of the scale of deployment and datasets. In essence, shuffle optimization is making a trade-off between scalability, performance, and fault tolerance. Riffle and Sailfish achieves better scalability by introducing an additional pass of the on-disk shuffle files, while the ThemisMR system aggregates the shuffle data in

memory. Riffle achieves better scalability and can process PB-level data, but incurs merge operation overhead.

Systematic measurement should be done to demonstrate and compare their performance under different use cases. While Riffle has been deployed and running efficiently in Facebook's highly-scalable Spark platform, we cannot assume that every deployment of big-data frameworks is with thousands of machines and processing petabytes of data. For a medium to small deployment, it will be helpful to compare various performance metrics, and provide guidelines to help users choose and configure the systems to best fit their specific workloads and resources.

5.3 Concluding Remarks

Advanced data analytics queries are becoming increasingly important to gain deep insights and drive crucial decisions from large volumes of data. The rapidly growing data, the complexity of analytics, and the various query requirements present an unprecedented challenge to big-data systems with only limited resources. With the recent development of data acquisition (such as Internet-of-Things devices) and processing techniques (such as deep learning), the data volume and complexity will likely continue to increase.

This dissertation argues that efficient and intelligent resource management can significantly improve the performance of advanced data analytics. By leveraging the application specific characteristics, the schedulers can make well-informed resource allocation decisions targeting the cluster-wide system performance. In addition to the demonstrated use cases, future big-data systems will soon see various new types of data and workloads. We believe the design principles are broadly applicable and will carry over to new scenarios and applications to conquer the resource management challenges.

Bibliography

[1] Apache Flink. https://flink.apache.org/.

[2] Apache Hadoop. Retrieved 02/08/2017, URL: http://hadoop.apache.org.

[3] Apache Hadoop NextGen MapReduce (YARN). https://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/YARN.html.

[4] Apache Ignite. https://ignite.apache.org/.

[5] Apache Spark. http://spark.apache.org/.

[6] Apache Spark Performance Tuning—Degree of Parallelism. https://tinyurl.com/y93l4lbu.

[7] Apache Spark @Scale: A 60 TB+ Production Use Case. https://code.facebook.com/posts/1671373793181703/.

[8] Apache Spark the Fastest Open Source Engine for Sorting a Petabyte. https://databricks.com/blog/2014/10/10/spark-petabyte-sort.html.

[9] Apache Storm. https://storm.apache.org/.

[10] Arimo TensorSpark. https://github.com/adatao/tensorspark.

[11] Associated Press Dataset - LDA. http://www.cs.columbia.edu/~blei/lda-c/.

[12] Avigilon. http://avigilon.com/products/.

[13] Azure Instances. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/.

[14] Caffe2. https://github.com/caffe2/caffe2.

[15] Facebook Disaggregate: Networking recap. https://code.facebook.com/posts/1887543398133443/.

[16] Facebook's Disaggregate Storage and Compute for Map/Reduce. https://tinyurl.com/ycewlve7.

[17] Genetec. https://www.genetec.com/.

[18] H2O: Open Source Platform for AI. https://docs.h2o.ai.

[19] Hadoop Capacity Scheduler. https://hadoop.apache.org/docs/r2.4.1/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html.

[20] Hadoop Fair Scheduler. https://hadoop.apache.org/docs/r2.4.1/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.

[21] LibSVM Data. https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/.

[22] Linux Containers LXC Introduction. https://linuxcontainers.org/lxc/introduction/.

[23] Live Video Analytics - Microsoft Research. https://www.microsoft.com/en-us/research/project/live-video-analytics/.

[24] LZ4: Extremely Fast Compression Algorithm. http://www.lz4.org.

[25] MapReduce-4049: Plugin for Generic Shuffle Service. https://issues.apache.org/jira/browse/MAPREDUCE-4049.

[26] Million Song Dataset. https://labrosa.ee.columbia.edu/millionsong/.

[27] MNIST Database. http://yann.lecun.com/exdb/mnist/.

[28] Netflix Prize). https://www.netflixprize.com/.

[29] Open ALPR. http://www.openalpr.com.

[30] OpenCV Documentation: Introduction to SIFT (Scale-Invariant Feature Transform). http://docs.opencv.org/3.1.0/da/df5/tutorial_py_sift_intro.html.

[31] OpenCV Documentation: Introduction to SURF (Speeded-Up Robust Features). http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_surf_intro/py_surf_intro.html.

[32] PASCAL Challenge 2008. http://largescale.ml.tu-berlin.de/instructions/.

[33] PyTorch. http://pytorch.org/.

[34] Snappy: A Fast Compressor/Decompressor. https://google.github.io/snappy/.

[35] Spark Configuration: External Shuffle Service. https://spark.apache.org/docs/latest/job-scheduling.html.

[36] SR 520 Bridge Tolling, WA. https://www.wsdot.wa.gov/Tolling/520/default.htm.

[37] Tim Sort. http://wiki.c2.com/?TimSort.

[38] Turnpike Enterprise Toll-by-Plate, FL. https://www.tollbyplate.com/index.

[39] Windows Job Objects. https://msdn.microsoft.com/en-us/library/windows/desktop/ms684161(v=vs.85).aspx.

[40] Working with Apache Spark. https://tinyurl.com/yaekw6rm.

[41] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-scale Machine Learning. In *USENIX OSDI*, 2016.

[42] Abadi, Daniel J and others. The Design of the Borealis Stream Processing Engine. In *CIDR*, Jan. 2005.

[43] S. Agarwal, B. Mozafari, A. Panda, Milner H., S. Madden, and I. Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *ACM EuroSys*, Apr. 2013.

[44] Tyler Akidau et al. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-order Data Processing. *VLDB*, Aug. 2015.

[45] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *USENIX NSDI*, 2017.

[46] Lisa Amini, Navendu Jain, Anshul Sehgal, Jeremy Silber, and Olivier Verscheure. Adaptive Control of Extreme-Scale Stream Processing Systems. In *IEEE ICDCS*, July 2006.

[47] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective Straggler Mitigation: Attack of the Clones. In *USENIX NSDI*, 2013.

[48] Ganesh Ananthanarayanan, Michael Chien-Chun Hung, Xiaoqi Ren, Ion Stoica, Adam Wierman, and Minlan Yu. GRASS: Trimming Stragglers in Approximation Analytics. In *USENIX NSDI*, Apr. 2014.

[49] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the Outliers in Map-reduce Clusters Using Mantri. In *USENIX OSDI*, 2010.

[50] Michael Anderson, Dolan Antenucci, Victor Bittorf, Matthew Burgess, Michael Cafarella, Arun Kumar, Feng Niu, Yongjoo Park, Christopher Ré, and Ce Zhang. Brainwash: A Data System for Feature Engineering. In *CIDR*, 2013.

[51] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational Data Processing in Spark. In *ACM SIGMOD*, 2015.

[52] Alvin AuYoung, Amin Vahdat, and Alex C Snoeren. Evaluating the Impact of Inaccurate Information in Utility-Based Scheduling. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, Nov. 2009.

[53] Brian Babcock, Surajit Chaudhuri, and Gautam Das. Dynamic Sample Selection for Approximate Query Processing. In *ACM SIGMOD*, 2003.

[54] Brian Babcock, Mayur Datar, and Rajeev Motwani. Load Shedding for Aggregation Queries over Data Streams. In *IEEE ICDE*, Mar. 2004.

[55] David Barrett. One Surveillance Camera for Every 11 People in Britain, Says CCTV Survey. `https://tinyurl.com/y8t7j2s6`, 2013.

[56] Anton Beloglazov and Rajkumar Buyya. Energy Efficient Resource Management in Virtualized Cloud Data Centers. In *IEEE CCGRID*, May 2010.

[57] Arie Ben-David and Eibe Frank. Accuracy of Machine Learning Models Versus "Hand Crafted" Expert Systems – A Credit Scoring Case Study. *Expert Systems with Applications*, 36(3):5264–5271, Apr. 2009.

[58] Josep Lluís Berral, Nicolas Poggi, David Carrera, Aaron Call, Rob Reinauer, and Daron Green. ALOJA-ML: A Framework for Automating Characterization and Knowledge Discovery in Hadoop Deployments. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2015.

[59] Arka Bhattacharya, David Culler, Eric Friedman, Ali Ghodsi, Scott Shenker, and Ion Stoica. Hierarchical Scheduling for Diverse Datacenter Workloads. In *ACM SoCC*, Nov. 2013.

[60] Jim Blythe. Visual Exploration and Incremental Utility Elicitation. In *AAAI*, July 2002.

[61] Norman Bobroff, Andrzej Kochut, and Kirk Beaty. Dynamic Placement of Virtual Machines for Managing SLA Violations. In *IFIP/IEEE International Symposium on Integrated Network Management*, 2007.

[62] LÃl'on Bottou and Olivier Bousquet. The Tradeoffs of Large Scale Learning. In *NIPS*, 2008.

[63] N. Boumal, P.-A. Absil, and C. Cartis. Global Rates of Convergence for Nonconvex Optimization on Manifolds. *ArXiv e-prints*, abs/1605.08101, May 2016.

[64] Craig Boutilier, Relu Patrascu, Pascal Poupart, and Dale Schuurmans. Regret-based Utility Elicitation in Constraint-based Decision Problems. In *IJCAI*, 2005.

[65] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *USENIX OSDI*, 2014.

[66] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[67] Brad Calder et al. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *ACM SOSP*, 2011.

[68] Don Carney, Uğur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. Operator Scheduling in a Data Stream Manager. In *VLDB*, 2003.

[69] Don Carney, UÇğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring Streams: a New Class of Data Management Applications. In *VLDB*, 2002.

[70] Urszula Chajewska, Daphne Koller, and Ronald Parr. Making Rational Decisions Using Adaptive Utility Elicitation. In *AAAI*, 2000.

[71] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John Wernsing, and DeLine Rob. Trill: A High-Performance Incremental Query Processor for Diverse Analytics. In *USENIX NSDI*, 2014.

[72] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *ArXiv e-prints*, abs/1512.01274, 2015.

[73] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stan Zdonik. Scalable Distributed Stream Processing. In *CIDR*, Jan. 2003.

[74] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *USENIX OSDI*, Broomfield, CO, 2014.

[75] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce Online. In *USENIX NSDI*, 2010.

[76] Emilio Coppa and Irene Finocchi. On Data Skewness, Stragglers, and MapReduce Progress Indicators. In *ACM SoCC*, 2015.

[77] Graham Cormode, Minos Garofalakis, Peter J. Haas, and Chris Jermaine. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Foundations and Trends in Databases*, 4(1-8211;3), Jan. 2012.

[78] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Exploiting Bounded Staleness to Speed Up Big Data Analytics. In *USENIX ATC*, 2014.

[79] Carlo Curino, Djellel E. Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. Reservation-based Scheduling: If You're Late Don't Blame Us! Nov. 2014.

[80] George B. Dantzig. Discrete-Variable Extremum Problems. *Operations Research 5(2): 266âĂŞ288*, 1957.

[81] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *USENIX OSDI*, 2004.

[82] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR*, 2009.

[83] Cliff Engle, Antonio Lupher, Reynold Xin, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: Fast Data Analysis Using Coarse-grained Distributed Memory. In *ACM SIGMOD*, 2012.

[84] Andrew D Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *ACM EuroSys*, 2012.

[85] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network Requirements for Resource Disaggregation. In *USENIX OSDI*, 2016.

[86] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *ACM SOSP*, 2003.

[87] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *USENIX NSDI*, 2011.

[88] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Choosy: Proportional Sharing for Datacenter Jobs with Constraints. In *ACM EuroSys*, 2013.

[89] 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 *USENIX OSDI*, 2014.

[90] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

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

[92] Laura M. Grupp, John D. Davis, and Steven Swanson. The Bleak Future of NAND Flash Memory. In *USENIX FAST*, 2012.

[93] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. MCDNN: An Approximation-Based Execution Framework for Deep Stream Processing Under Resource Constraints. In *ACM MobiSys*, 2016.

[94] Song Han, Huizi Mao, and William J. Dally. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *ArXiv e-prints*, abs/1510.00149, Oct. 2015.

[95] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference and Prediction*. Springer, 2nd edition, 2009.

[96] Joseph M Hellerstein, Peter J Haas, and Helen J Wang. Online Aggregation. In *ACM SIGMOD*, 1997.

[97] Adolfo Hernando, Ricardo Sanz, and R Calinescu. A Model-Based Approach to the Autonomic Management of Mobile Robot Resources. In *International Conference on Adaptive and Self-Adaptive Systems and Applications*, 2010.

[98] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A Self-tuning System for Big Data Analytics. In *CIDR*, pages 261–272, 2011.

[99] Frank Hersey. China to have 626 million surveillance cameras within 3 years. `https://tinyurl.com/ycgsyy9w`, 2017.

[100] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *USENIX NSDI*, 2011.

[101] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodík, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. Focus: Querying large video datasets with low latency and low cost. *ArXiv e-prints*, abs/1801.03493, 2018.

[102] Qi Huang, Petchean Ang, Peter Knowles, Tomasz Nykiel, Iaroslav Tverdokhlib, Amit Yajurvedi, Paul Dapolito VI, Xifan Yan, Maxim Bykov, Chuen Liang, Mohit Talwar, Abhishek Mathur, Sachin Kulkarni, Matthew Burke, and Wyatt Lloyd. SVE: Distributed Video Processing at Facebook Scale. In *ACM SOSP*, 2017.

[103] Chien-Chun Hung, Leana Golubchik, and Minlan Yu. Scheduling Jobs Across Geo-distributed Datacenters. In *ACM SoCC*, 2015.

[104] David E Irwin, Laura E Grit, and Jeffrey S Chase. Balancing Risk and Reward in a Market-Based Task Service. In *IEEE International Symposium on High Performance Distributed Computing*, 2004.

[105] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *ACM EuroSys*, 2007.

[106] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *ACM SOSP*, 2009.

[107] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population Based Training of Neural Networks. *ArXiv e-prints*, abs/1711.09846.

[108] J.M. Jaffe. Bottleneck Flow Control. *IEEE Transactions on Communications*, 29(7):954–962, 1981.

[109] S. Jain, V. Nguyen, M. Gruteser, and P. Bahl. Panoptes: Servicing Multiple Applications Simultaneously Using Steerable Cameras. In *16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, Apr. 2017.

[110] E. Douglas Jensen, Peng Li, and Binoy Ravindran. On Recent Advances in Time/Utility Function Real-Time Scheduling and Resource Management. *IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, 2005.

[111] Chris Jermaine, Subramanian Arumugam, Abhijit Pol, and Alin Dobra. Scalable Approximate Query Processing with the DBO Engine. *ACM Transactions on Database Systems*, 33(4):23, 2008.

[112] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *ACM International Conference on Multimedia*, 2014.

[113] Ramesh Johari and John N Tsitsiklis. Efficiency Loss in a Network Resource Allocation Game. *Mathematics of Operations Research*, 29(3):407–435, 2004.

[114] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Íñigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *USENIX OSDI*, 2016.

[115] S. Kambhampati, J. Kelley, C. Stewart, W. C. L. Stewart, and R. Ramnath. Managing Tiny Tasks for Data-Parallel, Subsampling Workloads. In *2014 IEEE International Conference on Cloud Engineering*, 2014.

[116] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. NoScope: Optimizing Neural Network Queries over Video at Scale. *VLDB*, 10(11), Aug. 2017.

[117] Vamsee Kasavajhala. Solid State Drive vs. Hard Disk Drive Price and Performance Study: A Dell Technical White Paper. *Dell PowerVault Storage Systems*, May 2011.

[118] Soila Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. An Analysis of Traces from a Production MapReduce Cluster. In *IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, 2010.

[119] Frank P Kelly, Aman K Maulloo, and David KH Tan. Rate Control for Communication Networks: Shadow Prices, Proportional Fairness and Stability. *Journal of the Operational Research Society*, 49(3):237–252, 1998.

[120] Jeffrey O Kephart. Research Challenges of Autonomic Computing. In *ACM ICSE*, 2005.

[121] Matej Kristan, Jiri Matas, Ales Leonardis, Michael Felsberg, Luka Cehovin, Gustavo Fernandez, Tomas Vojir, Gustav Hager, Georg Nebehay, and Roman Pflugfelder. The Visual Object Tracking (VOT) Challenge Results. In *IEEE ICCV Workshops*, Dec. 2015.

[122] Vibhore Kumar, Brian F Cooper, and Karsten Schwan. Distributed Stream Management Using Utility-Driven Self-Adaptive Middleware. In *IEEE ICAC*, 2005.

[123] S. Lacoste-Julien. Convergence Rate of Frank-Wolfe for Non-Convex Objectives. *ArXiv e-prints*, abs/1607.00345, July 2016.

[124] Quoc V. Le, Rajat Monga, Matthieu Devin, Greg Corrado, Kai Chen, Marc'Aurelio Ranzato, Jeffrey Dean, and Andrew Y. Ng. Building High-Level Features Using Large Scale Unsupervised Learning. *ArXiv e-prints*, abs/1112.6209, 2011.

[125] Yann LeCun, John S. Denker, and Sara A. Solla. Optimal Brain Damage. In *NIPS*. 1990.

[126] Ron Levy, Jay Nagarajarao, Giovanni Pacifici, Mike Spreitzer, Asser Tantawi, and Alaa Youssef. Performance Management for Cluster Based Web Services. In *Integrated Network Management VIII*, pages 247–261. Springer, 2003.

[127] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *ACM SoCC*, 2014.

[128] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Efficient Hyperparameter Optimization and Infinitely Many Armed Bandits. *ArXiv e-prints*, abs/1603.06560.

[129] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling Distributed Machine Learning with the Parameter Server. In *USENIX OSDI*, 2014.

[130] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *ACM ISCA*, 2009.

[131] Wei Lin, Zhengping Qian, Junwei Xu, Sen Yang, Jingren Zhou, and Lidong Zhou. StreamScope: Continuous Reliable Distributed Processing of Big Data Streams. In *USENIX NSDI*, Mar. 2016.

[132] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. Don't Get Caught in the Cold, Warm-up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems. In *USENIX OSDI*, Savannah, GA, 2016.

[133] Steven H Low and David E Lapsley. Optimization Flow Control-I: Basic Algorithm and Convergence. *IEEE/ACM Transactions on Networking*, 7(6):861–874, 1999.

[134] Dougal Maclaurin, David Duvenaud, and Ryan P. Adams. Gradient-based Hyperparameter Optimization through Reversible Learning. 2015.

[135] S. T. Maguluri, R. Srikant, and L. Ying. Stochastic Models of Load Balancing and Scheduling in Cloud Computing Clusters. In *IEEE INFOCOM*, 2012.

[136] Peter Marbach. Priority Service and Max-Min Fairness. In *IEEE INFOCOM*, 2002.

[137] M. D. McKay, R. J. Beckman, and W. J. Conover. A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code. *Technometrics*, 42(1):55–61, Feb. 2000.

[138] Xiangrui Meng, Joseph K. Bradley, Burak Yavuz, Evan R. Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. MLlib: Machine Learning in Apache Spark. *ArXiv e-prints*, abs/1505.06807, 2015.

[139] Robert C. Merton. *Continuous-Time Finance*. Blackwell, 1990.

[140] Dorian Minarolli and Bernd Freisleben. Utility-Based Resource Allocation for Virtual Machines in Cloud Computing. In *IEEE Symposium on Computers and Communications*, pages 410–417, 2011.

[141] Michael Mitzenmacher. The Power of Two Choices in Randomized Load Balancing. 12(10):1094–1104, Oct. 2001.

[142] Manfred Morari and Jay H Lee. Model Predictive Control: Past, Present and Future. *Computers & Chemical Engineering*, 23(4):667–682, 1999.

[143] Philipp Moritz, Robert Nishihara, Ion Stoica, and Michael I. Jordan. SparkNet: Training Deep Networks in Spark. *ArXiv e-prints*, abs/1511.06051, 2015.

[144] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query Processing, Resource Management, and Approximation in a Data Stream Management System. In *CIDR*, 2003.

[145] Hyeonseob Nam and Bohyung Han. Learning Multi-Domain Convolutional Neural Networks for Visual Tracking. *ArXiv e-prints*, abs/1510.07945, 2015.

[146] Karl Ni, Roger A. Pearce, Kofi Boakye, Brian Van Essen, Damian Borth, Barry Chen, and Eric X. Wang. Large-Scale Deep Learning on the YFCC100M Dataset. *ArXiv e-prints*, abs/1502.03409, 2015.

[147] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks. In *ACM SOSP*, 2017.

[148] Kay Ousterhout, Aurojit Panda, Joshua Rosen, Shivaram Venkataraman, Reynold Xin, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The Case for Tiny Tasks in Compute Clusters. Santa Ana Pueblo, NM, 2013.

[149] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making Sense of Performance in Data Analytics Frameworks. In *USENIX NSDI*, 2015.

[150] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, Low Latency Scheduling. In *ACM SOSP*, 2013.

[151] Pradeep Padala, Kang G Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, and Kenneth Salem. Adaptive Control of Virtualized Resources in Utility Computing Environments. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 289–302, 2007.

[152] Rina Panigrahy, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. Heuristics for Vector Bin Packing. In *Microsoft Research Technical Report*, Jan. 2011.

[153] Niketan Pansare, Vinayak R. Borkar, Chris Jermaine, and Tyson Condie. Online Aggregation for Large MapReduce Jobs. 4(11), 2011.

[154] D.M.W. Powers. Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness & Correlation. *Journal of Machine Learning Technologies*, 2(1):37–63, 2011.

[155] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. Low Latency Geo-distributed Data Analytics. In *ACM SIGCOMM*, 2015.

[156] Ariel Rabkin, Matvey Arye, Siddhartha Sen, Vivek Pai, and Michael J. Freedman. Aggregation and Degradation in JetStream: Streaming Analytics in the Wide Area. In *USENIX NSDI*, 2014.

[157] Sriram Rao, Raghu Ramakrishnan, Adam Silberstein, Mike Ovsiannikov, and Damian Reeves. Sailfish: A Framework for Large Scale Data Processing. In *ACM SoCC*, 2012.

[158] A. Rasmussen, M. Conley, R. Kapoor, V.T. Lam, G. Porter, and A. Vahdat. ThemisMR: An I/O-efficient MapReduce. *Technical Report (University of California, San Diego. Department of Computer Science and Engineering)*, 2012.

[159] Alexander Rasmussen, Vinh The Lam, Michael Conley, George Porter, Rishi Kapoor, and Amin Vahdat. Themis: An I/O-efficient MapReduce. In *ACM SoCC*, 2012.

[160] Alexander Rasmussen, George Porter, Michael Conley, Harsha V. Madhyastha, Radhika Niranjan Mysore, Alexander Pucher, and Amin Vahdat. TritonSort: A Balanced Large-scale Sorting System. In *USENIX NSDI*, 2011.

[161] Brian T Ratchford. Cost-Benefit Models for Explaining Consumer Choice and Information Seeking Behavior. *Management Science*, 28(2):197–212, Feb. 1982.

[162] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2nd edition, 2003.

[163] Frank Seide and Amit Agarwal. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *KDD*, 2016.

[164] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[165] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *ArXiv e-prints*, abs/1409.1556, 2014.

[166] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical Bayesian Optimization of Machine Learning Algorithms. In *NIPS*, Dec. 2012.

[167] Evan R. Sparks, Ameet Talwalkar, Daniel Haas, Michael J. Franklin, Michael I. Jordan, and Tim Kraska. Automating Model Search for Large Scale Machine Learning. In *ACM SoCC*, 2015.

[168] Malgorzata Steinder, Ian Whalley, David Carrera, Ilona Gaweda, and David Chess. Server Virtualization in Autonomic Management of Heterogeneous Workloads. In *IFIP/IEEE International Symposium on Integrated Network Management*, 2007.

[169] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Nikolas Ioannou, and Ioannis Koltsidas. Crail: A High-Performance I/O Architecture for Distributed Data Processing. *IEEE Data Eng. Bull.*, 40(1):38–49, 2017.

[170] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load Shedding in a Data Stream Manager. In *VLDB*, 2003.

[171] Nesime Tatbul, UÇğur Çetintemel, and Stan Zdonik. Staying Fit: Efficient Load Shedding Techniques for Distributed Stream Processing. In *VLDB*, 2007.

[172] Gerald Tesauro, Rajarshi Das, William E Walsh, and Jeffrey O Kephart. Utility-Function-Driven Resource Allocation in Autonomic Systems. In *ICAC*, 2005.

[173] Thomas N. Theis and H. S. Philip Wong. The End of Moore's Law: A New Beginning for Information Technology. *Computing in Science and Engg.*, 19(2):41–50, Mar. 2017.

[174] Yoh'ichi Tohkura. A Weighted Cepstral Distance Measure for Speech Recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 35:1414–1422, 1987.

[175] Yi-Cheng Tu, Mohamed Hefeeda, Yuni Xia, Sunil Prabhakar, and Song Liu. Control-Based Quality Adaptation in Data Stream Management Systems. In *Database and Expert Systems Applications*, 2005.

[176] Yi-Cheng Tu, Song Liu, Sunil Prabhakar, and Bin Yao. Load Shedding in Stream Databases: a Control-Based Approach. In *VLDB*, 2006.

[177] C. J Van Rijsbergen. Information Retrieval. *Butterworth, 2nd edition*, 1979.

[178] Shivaram Venkataraman, Aurojit Panda, Ganesh Ananthanarayanan, Michael J. Franklin, and Ion Stoica. The Power of Choice in Data-aware Cluster Scheduling. In *USENIX OSDI*, 2014.

[179] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *USENIX NSDI*, Santa Clara, CA, 2016.

[180] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. ARIA: Automatic Resource Inference and Allocation for Mapreduce Environments. In *ICAC*, 2011.

[181] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *ACM EuroSys*, 2015.

[182] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. Characterizing Cloud Computing Hardware Reliability. In *ACM SoCC*, 2010.

[183] Ashish Vulimiri, Carlo Curino, P. Brighten Godfrey, Thomas Jungblut, Jitu Padhye, and George Varghese. Global Analytics in the Face of Bandwidth and Regulatory Constraints. In *USENIX NSDI*, Oakland, CA, 2015.

[184] Ernesto Wandeler and Lothar Thiele. Real-Time Interfaces for Interface-Based Design of Real-Time Systems with Fixed Priority Scheduling. In *Proceedings of the 5th ACM International Conference on Embedded Software*, pages 80–89, 2005.

[185] Y. Wang, R. Goldstone, W. Yu, and T. Wang. Characterization and Optimization of Memory-Resident MapReduce on HPC Systems. In *IEEE 28th International Parallel and Distributed Processing Symposium*, 2014.

[186] Yandong Wang, Xinyu Que, Weikuan Yu, Dror Goldenberg, and Dhiraj Sehgal. Hadoop Acceleration Through Network Levitated Merge. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[187] Yuan Wei, Vibha Prasad, Sang H Son, and John A Stankovic. Prediction-Based QoS Management for Real-Time Data Streams. In *IEEE RTSS*, 2006.

[188] Caesar Wu and Rajkumar Buyya. *Cloud Data Centers and Cost Modeling: A Complete Guide To Planning, Designing and Building a Cloud Data Center*. Morgan Kaufmann Publishers Inc., 1st edition, 2015.

[189] Wencong Xiao, Jilong Xue, Youshan Miao, Zhen Li, Cheng Chen, Ming Wu, Wei Li, and Lidong Zhou. Tux2: Distributed Graph Computation for Machine Learning. In *USENIX NSDI*, 2017.

[190] Tao Ye and Shivkumar Kalyanaraman. A Recursive Random Search Algorithm for Large-scale Network Parameter Configuration. 2003.

[191] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *USENIX NSDI*, 2012.

[192] Matei Zaharia, Tathagatha Das, Haoyuan Li, Tim Hunter, Scott Shenker, and Ion Stoica. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *ACM SOSP*, 2013.

[193] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *USENIX OSDI*, 2008.

[194] Kai Zeng, Sameer Agarwal, and Ion Stoica. iOLAP: Managing Uncertainty for Efficient Incremental OLAP. In *ACM SIGMOD*, 2016.

[195] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. Live Video Analytics at Scale with Approximation and Delay-Tolerance. In *USENIX NSDI*, 2017.

[196] Haoyu Zhang, Brian Cho, Ergin Seyfe, Avery Ching, and Michael J. Freedman. Riffle: Optimized Shuffle Service for Large-Scale Data Analytics. In *ACM EuroSys*, 2018.

[197] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J. Freedman. SLAQ: Quality-Driven Scheduling for Distributed Machine Learning. In *ACM SoCC*, 2017.

[198] Tan Zhang, Aakanksha Chowdhery, Paramvir Bahl, Kyle Jamieson, and Suman Banerjee. The Design and Implementation of a Wireless Video Surveillance System. In *ACM MobiCom*, 2015.

[199] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning. In *ACM SoCC*, Santa Clara, CA, 2017.

[200] Shlomo Zilberstein. Using anytime algorithms in intelligent systems. *AI magazine*, 17(3):73, 1996.