System Design for Large Scale Machine Learning

By

Shivaram Venkataraman

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:
Professor Michael J. Franklin, Co-chair
Professor Ion Stoica, Co-chair
Professor Benjamin Recht
Professor Ming Gu

Fall 2017

System Design for Large Scale Machine Learning

Abstract

System Design for Large Scale Machine Learning

by

Shivaram Venkataraman

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Michael J. Franklin, Co-chair

Professor Ion Stoica, Co-chair

The last decade has seen two main trends in the large scale computing: on the one hand we have seen the growth of cloud computing where a number of big data applications are deployed on shared cluster of machines. On the other hand there is a deluge of machine learning algorithms used for applications ranging from image classification, machine translation to graph processing, and scientific analysis on large datasets. In light of these trends, a number of challenges arise in terms of how we program, deploy and achieve high performance for large scale machine learning applications.

In this dissertation we study the execution properties of machine learning applications and based on these properties we present the design and implementation of systems that can address the above challenges. We first identify how choosing the appropriate hardware can affect the performance of applications and describe Ernest, an efficient performance prediction scheme that uses experiment design to minimize the cost and time taken for building performance models. We then design scheduling mechanisms that can improve performance using two approaches: first by improving data access time by accounting for locality using data-aware scheduling and then by using scalable scheduling techniques that can reduce coordination overheads.

To my parents

Contents

List of Figures v

List of Tables viii

Acknowledgments ix

1 Introduction 1
1.1 Machine Learning Workload Properties 2
1.2 Cloud Computing: Hardware & Software 2
1.3 Thesis Overview 3
1.4 Organization 4

2 Background 6
2.1 Machine Learning Workloads 6
2.1.1 Empirical Risk Minimization 6
2.1.2 Iterative Solvers 7
2.2 Execution Phases 9
2.3 Computation Model 11
2.4 Related Work 12
2.4.1 Cluster scheduling 13
2.4.2 Machine learning frameworks 13
2.4.3 Continuous Operator Systems 13
2.4.4 Performance Prediction 14
2.4.5 Database Query Optimization 14
2.4.6 Performance optimization, Tuning 15

3 Modeling Machine Learning Jobs 16
3.1 Performance Prediction Background 17
3.1.1 Performance Prediction 17
3.1.2 Hardware Trends 18
3.2 Ernest Design 20
3.2.1 Features for Prediction 21
3.2.2 Data collection 22

3.2.3 Optimal Experiment Design 22
3.2.4 Model extensions 24
3.3 Ernest Implementation 25
3.3.1 Job Submission Tool 25
3.3.2 Handling Sparse Datasets 26
3.3.3 Straggler mitigation by over-allocation 26
3.4 Ernest Discussion 27
3.4.1 Model reuse 27
3.4.2 Using Per-Task Timings 28
3.5 Ernest Evaluation 28
3.5.1 Workloads and Experiment Setup 29
3.5.2 Accuracy and Overheads 29
3.5.3 Choosing optimal number of instances 31
3.5.4 Choosing across instance types 31
3.5.5 Experiment Design vs. Cost-based 33
3.5.6 Model Extensions 34
3.6 Ernest Conclusion 34

4 Low-Latency Scheduling **35**
4.1 Case for low-latency scheduling 36
4.2 Drizzle Design 37
4.2.1 Group Scheduling 37
4.2.2 Pre-Scheduling Shuffles 39
4.2.3 Adaptivity in Drizzle 39
4.2.4 Automatically selecting group size 40
4.2.5 Conflict-Free Shared Variables 41
4.2.6 Data-plane Optimizations for SQL 41
4.2.7 Drizzle Discussion 43
4.3 Drizzle Implementation 44
4.4 Drizzle Evaluation 45
4.4.1 Setup 45
4.4.2 Micro benchmarks 45
4.4.3 Machine Learning workloads 47
4.4.4 Streaming workloads 49
4.4.5 Micro-batch Optimizations 51
4.4.6 Adaptivity in Drizzle 52
4.5 Drizzle Conclusion 53

5 Data-aware scheduling **54**
5.1 Choices and Data-Awareness 55
5.1.1 Application Trends 55
5.1.2 Data-Aware Scheduling 55

5.1.3 Potential Benefits 58
5.2 Input Stage 58
5.2.1 Choosing any K out of N blocks 58
5.2.2 Custom Sampling Functions 59
5.3 Intermediate Stages 60
5.3.1 Additional Upstream Tasks 60
5.3.2 Selecting Best Upstream Outputs 62
5.3.3 Handling Upstream Stragglers 63
5.4 KMN Implementation 65
5.4.1 Application Interface 66
5.4.2 Task Scheduling 66
5.4.3 Support for extra tasks 67
5.5 KMN Evaluation 68
5.5.1 Setup 69
5.5.2 Benefits of KMN 69
5.5.3 Input Stage Locality 72
5.5.4 Intermediate Stage Scheduling 72
5.6 KMN Conclusion 75

6 Future Directions & Conclusion **76**
6.1 Future Directions 77
6.2 Concluding Remarks 78

Bibliography **79**

List of Figures

2.1 Execution of a machine learning pipeline used for text analytics. The pipeline consists of featurization and model building steps which are repeated for many iterations. . . . 9
2.2 Execution DAG of a machine learning pipeline used for speech recognition. The pipeline consists of featurization and model building steps which are repeated for many iterations. 10
2.3 Execution of Mini-batch SGD and Block coordinate descent on a distributed runtime. . 11
2.4 Execution of a job when using the batch processing model. We show two iterations of execution here. The left-hand side shows the various steps used to coordinate execution. The query being executed in shown on the right hand side 12

3.1 Memory bandwidth and network bandwidth comparison across instance types 17
3.2 Scaling behaviors of commonly found communication patterns as we increase the number of machines. 19
3.3 Performance comparison of a Least Squares Solver (LSS) job and Matrix Multiply (MM) across similar capacity configurations. 20
3.4 Comparison of different strategies used to collect training data points for KMeans. The labels next to the data points show the (number of machines, scale factor) used. 24
3.5 CDF of maximum number of non-zero entries in a partition, normalized to the least loaded partition for sparse datasets. 26
3.6 CDFs of STREAM memory bandwidths under four allocation strategies. Using a small percentage of extra instances removes stragglers. 26
3.7 Running times of GLM and Naive Bayes over a 24-hour time window on a 64-node EC2 cluster. 26
3.8 Prediction accuracy using Ernest for 9 machine learning algorithms in Spark MLlib. . . 28
3.9 Prediction accuracy for GenBase, TIMIT and Adam queries. 28
3.10 Training times vs. accuracy for TIMIT and MLlib Regression. Percentages with respect to actual running times are shown. 30
3.11 Time per iteration as we vary the number of instances for the TIMIT pipeline and MLlib Regression. Time taken by actual runs are shown in the plot. 31
3.12 Time taken for 50 iterations of the TIMIT workload across different instance types. Percentages with respect to actual running times are shown. 32
3.13 Time taken for Sort and MarkDup workloads on ADAM across different instance types. 32

3.14 Ernest accuracy and model extension results. 33
3.15 Prediction accuracy improvements when using model extensions in Ernest. Workloads used include sparse GLM classification using KDDA, splice-site datasets and a random projection linear algebra job. 33

4.1 Breakdown of average time taken for task execution when running a two-stage `treeReduce` job using Spark. The time spent in scheduler delay and task transfer (which includes task serialization, deserialization, and network transfer) grows as we increase cluster size. 36
4.2 Group scheduling amortizes the scheduling overheads across multiple iterations of a streaming job. 38
4.3 Using pre-scheduling, execution of a iteration that has two stages: the first with 4 tasks; the next with 2 tasks. The driver launches all stages at the beginning (with information about where output data should be sent to) so that executors can exchange data without contacting the driver. 38
4.4 Micro-benchmarks for performance improvements from group scheduling and pre-scheduling. 45
4.5 Time taken per iteration of Stochastic Gradient Descent (SGD) run on the RCV1 dataset. We see that using sparse updates, Drizzle can scale better as the cluster size increases. 48
4.6 Latency and throughput comparison of Drizzle with Spark and Flink on the Yahoo Streaming benchmark. 49
4.7 Effect of micro-batch optimization in Drizzle in terms of latency and throughput. . . . 49
4.8 Behavior of Drizzle across streaming benchmarks and how the group size auto-tuning behaves for the Yahoo streaming benchmark. 50
4.9 Effect of varying group size in Drizzle. 51

5.1 Late binding allows applications to specify more inputs than tasks and schedulers dynamically choose task inputs at execution time. 56
5.2 Value of balanced network usage for a job with 4 map tasks and 4 reduce tasks. The left-hand side has unbalanced cross-rack links (maximum of 6 transfers, minimum of 2) while the right-hand side has better balance (maximum of 4 transfers, minimum of 3). 57
5.3 Cross-rack skew and input-stage locality simulation. 59
5.4 Probability of input-stage locality when using a sampling function which outputs f disjoint samples. Sampling functions specify additional constraints for samples. 59
5.5 Cross-rack skew as we vary M/K for uniform and log-normal distributions. Even 20% extra upstream tasks greatly reduces network imbalance for later stages. 61
5.6 CDF of cross-rack skew as we vary M/K for the Facebook trace. 61
5.7 Simulations to show how choice affects stragglers and downstream transfer 64
5.8 An example of a query in SQL, Spark and KMN 67
5.9 Execution DAG for Stochastic Gradient Descent (SGD). 69
5.10 Benefits from using KMN for Stochastic Gradient Descent 69

5.11 Comparing baseline and KMN-1.05 with sampling-queries from Conviva. Numbers on the bars represent percentage improvement when using KMN-$M/K = 1.05$. 70
5.12 Overall improvement from KMN compared to baseline. Numbers on the bar represent percentage improvement using KMN-$M/K = 1.05$. 71
5.13 Improvement due to memory locality for the Map Stage for the Facebook trace. Numbers on the bar represent percentage improvement using KMN-$M/K = 1.05$. 71
5.14 Job completion time and locality as we increase utilization. 72
5.15 Boxplot showing utilization distribution for different values of average utilization. . . . 72
5.16 Shuffle improvements when running extra tasks. 73
5.17 Difference in shuffle performance as cross-rack skew increases 73
5.18 Benefits from straggler mitigation and delayed stage launch. 74
5.19 CDF of % time that the job was delayed 75
5.20 CDF of % of extra map tasks used. 75
5.21 Difference between using greedy assignment of reducers versus using a round-robin scheme to place reducers among racks with upstream tasks. 75

List of Tables

3.1 Models built by Non-Negative Least Squares for MLlib algorithms using `r3.xlarge` instances. Not all features are used by every algorithm. 22
3.2 Cross validation metrics comparing different models for Sparse GLM run on the splice-site dataset. 24

4.1 Breakdown of aggregations used in a workload containing over 900,000 SQL and streaming queries. 42

5.1 Distribution of job sizes in the scaled down version of the Facebook trace used for evaluation. 68
5.2 Improvements over baseline, by job size and stage 69
5.3 Shuffle time improvements over baseline while varying M/K 73
5.4 Shuffle improvements with respect to baseline as cross-rack skew increases. 74

Acknowledgments

This dissertation would not have been possible without the guidance of my academic co-advisors Mike Franklin and Ion Stoica. Mike was the primary reason I started my PhD at UC Berkeley. From the first call he gave before visit day, to helping me find my research home in the AMPLab, to finally put together my job talk, Mike has been a constant guide in structuring my research in graduate school.

Though I entered Berkeley as a database student, Ion has been singly responsible for making me a successful systems researcher and I owe most of my skills on how to do research to Ion. Through my PhD and in the rest of my career I hope to follow his advice to focus on making impactful contributions.

It is not an exaggeration to say that my research career changed significantly after I started working with Ben Recht. All of my knowledge of machine learning comes from the time that Ben took to teach me. Ben's approach to research has been also helped me understand how to distill valuable research problems.

A number of other professors at Berkeley including Jim Demmel, Joseph Gonzales, Ming Gu, Joe Hellerstein, Randy Katz, Sylvia Ratnasamy, Scott Shenker took time to give me valuable feedback about my research.

I was one of the many systems graduate students that started in the same year and I was fortunate to be a part of this exceptionally talented group. Among them Kay Ousterhout, Aurojit Panda and Evan Sparks became my closest collaborators and even better friends. Given any situation Kay always knew what was the right question to ask and her quality of striving for perfection in everything, from CPU utilization to cookie recipes, continues to inspire me. Panda on the other hand always had the answer to any question I could come up with and his kindness to help under any circumstance helped me get through various situations in graduate school. Panda was also one of the main reasons I was able to go through the academic job process successfully. Evan Sparks had the happy knack of being interested in the same research problem of building systems for machine learning. Evan will always be the one I'll blame for introducing me to golf and along with Katie and Charlotte, he gave me a second home at Berkeley. Dan Haas provided humor in the cubicle, made a great cup of tea in the afternoons and somehow managed to graduate without co-authoring a paper. Justine Sherry made me a morning person and, along with the gang at 1044, hosted me on many Friday evening.

A number of students, post-docs and researchers in the AMPLab helped me crystallize my ideas through many research discussions. Ganesh Anathanarayanan, Ali Ghodsi and Matei Zaharia were great exemplars on doing good research and helped me become a better researcher. Stephen Tu, Rebecca Roelofs and Ashia Wilson spent many hours in front of whiteboards helping me understand various machine learning algorithms. Peter Bailis, Andrew Wang and Sara Alspaugh were valuable collaborators during my first few years.

The NetSys lab adopted me as a member even though I did no networking research (thanks Syliva and Scott!) and, Colin Scott, Radhika Mittal and Amin Tootoonchian graciously let me use their workspace. Kattt Atchley, Boban Zarkovich and Carlyn Chinen provided administrative help and Jon Kuroda made sure I never had an IT problem to worry about. Roy Campbell, Matthew Caesar, Partha Ranganathan, Niraj Tolia and Indrajit Roy introduced me to research during my Masters at UIUC and were instrumental in me applying for a PhD.

Finally, I'd like to thank all my family and friends for their encouragement. I would especially like to thank my parents for their constant support and for encouraging me to pursue my dreams.

Chapter 1

Introduction

Machine learning methods power the modern world with applications ranging from natural language processing [32], image classification [56] to genomics [158] and detecting supernovae [188] in astrophysics. The ubiquity of massive data [50] has made these algorithmic methods viable and accurate across a variety of domains [90, 118]. Supervised learning methods used to classify images are developed using millions of labeled images [107] while scientific methods like supernovae detection or solar flare detection [99] are powered by high resolution images continuously captured from telescopes.

To obtain high accuracy machine learning methods typically need to process large amounts of data [81]. For example, machine learning models used in applications like language modeling [102] are trained on a billion word datasets [45]. Similarly in scientific applications like genome sequencing [133] or astrophysics, algorithms need to process terabytes of data captured every day. The decline of Moore's law, where the processing speed of a single core no longer scales rapidly, and limited bandwidth to storage media like SSD or hard disks [67], makes it both inefficient and in some cases impossible to use a single machine to execute algorithms on large datasets. Thus there is a shift towards using distributed computing architectures where a number of machines are used in coordination to execute machine learning methods.

Using a distributed computing architecture has become especially prevalent with the advent of cloud computing [19], where users can easily provision a number of machines for a short time duration. Along with the limited duration, cloud computing providers like Amazon EC2 also allow users to choose the amount of memory, CPU and disk space provisioned. This flexibility makes cloud computing an attractive choice for running large scale machine learning methods. However there are a number of challenges in efficiently using large scale compute resources. These include questions on how the coordination or communication across machines is managed and how we can achieve high performance while remaining resilient to machine failures [184].

In this thesis, we focus on the design of systems used to execute large scale machine learning methods. To influence our design, we characterize the performance of machine learning methods when they are run on a cluster of machines and use this to develop systems that can improve performance and efficiency at scale. We next review the important trends that lead to the systems challenges at hand and present an overview of the key results developed in this thesis.

1.1 Machine Learning Workload Properties

Machine learning methods are broadly aimed at learning models from previously collected data and applying these models to new, unseen data. Thus machine learning methods typically consist of two main phases: a *training* phase where a model is built using training data and a *inference* phase where the model is applied. In this thesis we will focus only on the training of machine learning models.

Machine learning methods can be further classified into supervised and unsupervised methods. At a high level, supervised methods use labeled datasets where each input data item has a corresponding label. These methods can be used for applications like classifying an object into one of many classes. On the other hand, unsupervised methods typically operate on just the input data and can be used for applications like clustering where the number or nature of classes is not known beforehand. For supervised learning methods, having a greater amount of training data means that we can build a better model that can generate predictions with greater accuracy.

From a systems perspective large machine learning methods present a new workload class that has a number of unique properties when compared with traditional data processing workloads. The main properties we identify are:

- Machine learning algorithms are developed using linear algebra operations and hence are *computational and communication* intensive [61].
- Further, as the machine learning models assume that the training data has been sampled from a distribution [28], they build an model that *approximates* the best model on the distribution.
- A number of machine learning methods make incremental progress: i.e., the algorithms are initialized at random and at each *iteration* [29, 143], they make progress towards the final model.
- Iterative machine learning algorithms also have specific data access patterns where every iteration is based on a *sample* [115, 163] of the input data.

We provide examples on how these properties manifest in real world applications in Chapter 2.

The above properties both provide flexibility and impose constraints on systems used to execute machine learning methods. The resource usage change means that systems now need to be carefully architected to balance computation and communication. Further the iterative nature implies that I/O overhead and coordination per-iteration needs to be minimized. On the other hand, the fact that the models built are approximate and only use a sample of input data at each iteration provides system designers additional flexibility. We develop systems that exploit these properties in this thesis.

1.2 Cloud Computing: Hardware & Software

The idea of cloud computing, where users can allocate compute resources on demand, has brought to reality the long held dream of computing as a utility. Cloud computing also changes the cost model: instead of paying to own and maintain machines, users only pay for the time machines are used.

In addition to the cost model, cloud computing also changes the resource optimization goals for users. Traditionally users aimed to optimize the algorithms or software used given the fixed hardware that was available. However cloud computing providers like Amazon EC2 allow users to select how much memory, CPU, disk should be allocated per instance. For example on EC2 users could provision a `r3.8xlarge` instance with 16 cores, 244 GB of memory or a `c5.18xlarge` which has 36 cores, 144 GB of memory. These are just two examples out of more than fifty different instance types offered. The enormous flexibility offered by cloud providers means that it is now possible to jointly optimize both the resource configuration and the algorithms used.

With the widespread use of cluster computing, there have also been a number of systems developed to simplify large scale data processing. MapReduce [57] introduced a high level programming model where users could supply the computation while the system would take care of other concerns like handling machine failures or determining which computation would run on which machine. This model was further generalized to general purpose dataflow programs in systems like Dryad [91], DryadLINQ [182] and Spark [185]. These systems are all based on the bulk-synchronous parallel (BSP) model [166] where all the machines coordinate after completing one step of the computation.

However such general purpose frameworks are typically agnostic to the machine learning workload properties we discussed in the previous section. In this thesis we therefore look at how to design systems that can improve performance for machine learning methods while retaining properties like fault tolerance.

1.3 Thesis Overview

In this thesis we study the structure and property of machine learning applications from a systems perspective. To do this, we first survey a number of real world, large scale machine learning workloads and discuss how the properties we identified in Section 1.1 are relevant to system design. Based on these properties we then look at two main problems: performance modeling and task scheduling.

Performance Modeling: In order to improve performance, we first need to understand the performance of machine learning applications as the cluster and data sizes change. Traditional approaches that monitor repeated executions of a job [66], can make it expensive to build a performance model. Our main insight is that machine learning jobs have predictable structure in terms of computation and communication. Thus we can build performance models based on the behavior of the job on small samples of data and then predict its performance on larger datasets and cluster sizes. To minimize the time and resources spent in building a model, we use optimal experiment design [139], a statistical technique that allows us to collect as few training points as required. The performance models we develop can be used to both inform the deployment strategy and provide insight into how the performance is affected as we scale the data and cluster used.

Scheduling using ML workload properties: Armed with the performance model and the workload characteristics, we next study how we can improve performance for large scale machine learning workloads. We split performance into two major parts, the data-plane and control-plane, and we systematically study methods to improve the performance of each of them by making them aware of the properties of machine learning algorithms. To optimize the data plane, we design a data-aware scheduling mechanism that can minimize the amount of time spent in accessing data from disk or the network. To minimize the amount of time spent in coordination, we propose scheduling techniques that ensure low latency execution at scale.

Contributions: In summary, the main contributions of this thesis are

- We characterize large scale machine learning algorithms and present case studies on which properties of these workloads are important for system design.
- Using the above characterization we describe efficient techniques to build performance models that can accurately predict running time. Our performance models are useful to make deployment decisions and can also help users understand how the performance changes as the number and type of machines used change.
- Based on the the performance models we then describe how the scalability and performance of machine learning applications can be improved using scheduling techniques that exploit structural properties of the algorithms.
- Finally we present detailed performance evaluations on a number of benchmarks to quantify the benefits from each of our techniques.

1.4 Organization

This thesis incorporates our previously published work [168–170] and is organized as follows. Chapter 2 provides background on large scale machine learning algorithms and also surveys existing systems developed for scalable data processing.

Chapter 3 studies the problem of how we can efficiently deploy machine learning applications. The key to address this challenge is developing a performance prediction framework that can accurately predict the running time on a specified hardware configuration, given a job and its input. We develop Ernest [170], a performance prediction framework that can provide accurate predictions with low training overhead.

Following that Chapter 4 and Chapter 5 present new scheduling techniques that can improve performance at scale. Chapter 4 looks at how we can reduce the coordination overhead for low latency iterative methods while preserving fault tolerance. To do this we develop two main techniques: group scheduling and pre-scheduling. We build these techniques in a system called Drizzle [169] and also study how these techniques can be applied to other workloads like large scale stream processing. We also discuss how Drizzle can be used in conjunction with other systems like parameter servers, used for managing machine learning models, and compare the performance of our execution model to other widely used execution models.

We next study how to minimize data access latency for machine learning applications in Chapter 5. A number of machine learning algorithms process small subsets or samples of input data at each iteration and we exploit the number of choices available in this process to develop a data-aware scheduling mechanism in a system called KMN [168]. The KMN scheduler improves locality of data access and also minimizes the amount of data transferred across machines between stages of computation. We also extend KMN to study how other workloads like approximate query processing can be benefit from similar scheduling improvements.

Finally, Chapter 6 discusses directions for future research on systems for large scale learning and how some of the more recent trends in hardware and workloads could influence system design. We then conclude with a summary of the main results.

Chapter 2

Background

2.1 Machine Learning Workloads

We next study examples of machine learning workloads to characterize the properties that make them different from traditional data analytics workloads. We focus on supervised learning methods, where given training data and its corresponding labels, the ML algorithm learns a model that can predict labels on unseen data. Similar properties are also exhibited by unsupervised methods like K-Means clustering. Supervised learning algorithms that are used to build a model are typically referred to as optimization algorithms and these algorithms seek to minimize the error in the model built. One of the frameworks to analyze model error is Empirical Risk Minimization (ERM) and we next study how ERM can be used to understand properties of ML algorithms.

2.1.1 Empirical Risk Minimization

Consider a case where we are given n training data points $x_1 ... x_n$ and the corresponding labels $y_1 ... y_n$. We denote L as a loss function that returns how "close" a label is from the predicted label. Common loss functions include square distance for vectors or $0-1$ loss for binary classification. In this setup our goal is to learn a function f that minimizes the expected value of the loss. Assuming f belongs to a family of functions $\mathcal{F}$, optimization algorithms can be expressed as learning an model $\hat{f}$ which is defined as follows:

$$\mathbb{E}_n(f) = \frac{1}{n} \sum_{i=1}^{n} L(f(x_i), y_i) \tag{2.1}$$

$$\hat{f} = \arg\min_{f \in \mathcal{F}} \mathbb{E}_n(f) \tag{2.2}$$

The error in the model that is learned consists of three parts: the approximation error ε_{app}, the estimation error ε_{est} and the optimization error ε_{opt}. The approximation error comes from the function family that we choose to optimize over, while the estimation error comes from the fact that we only have a sample of input data for training. Finally the optimization error comes from the

Algorithm 1 Mini-batch SGD for quadratic loss. From [28,29]

```
Input: data $X \in \mathcal{X}^{n \times d}$, $Y \in R^{n \times k}$, number of iterations $n$, step size $s$
       mini-batch size $b \in \{1, ..., n\}$.
$\pi \leftarrow$ random permutation of $\{1, ..., n\}$.
$w \leftarrow 0_{d \times k}$.
for $i = 1$ to $n$ do
    $\pi \leftarrow$ random sample of size $b$ from $\{1, ..., n\}$.
    $X_b \leftarrow$ FEATUREBLOCK($X, \mathcal{I}_{\pi_i}$). /* Row block. */
    $\nabla f \leftarrow (X_b^{\mathsf{T}}(X_b * W)) - X_b^{\mathsf{T}}(Y)$.
    $W \leftarrow W - s * \nabla f$.
```

optimization algorithm. There are two main takeaways that we can see from this formulation. First is that as the number of data points available increases the estimation error should decrease, thereby showing why using more data often leads to better models. Second we see that the optimization error only needs to be on the same order as the other two sources of error. Hence if we are running an optimization algorithm it is good enough to use an *approximate method*.

For example if we were optimizing the square loss $\min_{W} ||XW - Y||_2$ then the exact solution is $W^* = (X^T X)^{-1}(X^T Y)$ where X and Y represent the data and label matrix respectively. If we assume the number of dimensions in the feature vector is d then it takes $O(nd^2) + O(d^3)$ to compute the exact solution. On the other hand as we only need an approximate solution we can use an iterative method like conjugate gradient or Gauss-Seidel that can give provide an approximate answer much faster. This also has implications for systems design as we can now have more flexibility in the our execution strategies while building models that are within the approximation bounds.

Next we look at two patterns used by iterative solvers and discuss the main factors that influence their performance.

2.1.2 Iterative Solvers

Consider an iterative solver whose input is a data matrix $X \in \mathbb{R}^{n \times d}$ and a label matrix $Y \in R^{n \times k}$. Here n represents the number of data points, d the dimension of each data point and k the dimension of the label. In such a case there are two ways in which iterative solvers proceed: at every iteration, they either sample a subset of the examples (rows) or sample a subset of the dimensions (columns) to construct a smaller problem. They then use the smaller problem to improve the current model and repeat this process until the model converges. There are two main characteristics we see here: first algorithms are *iterative* and run a large number of iterations to converge on a solution. Second algorithms *sample* a subset of the data at each iteration. We next look at two examples of iterative solvers that sample examples and dimensions respectively. For both cases we consider a square loss function

Mini-batch Gradient Descent: Let us first consider a mini-batch gradient descent algorithm shown in Algorithm 1. In this algorithm there is a specified batch size b that is provided and

Algorithm 2 BCD for quadratic loss. From [162]

```
Input: data $X \in \mathcal{X}^{n \times d}$, $Y \in R^{n \times k}$, number of epochs $n_e$,
       block size $b \in \{1, ..., d\}$.
$\pi \leftarrow$ random permutation of $\{1, ..., d\}$.
$\mathcal{I}_1, ..., \mathcal{I}_{\frac{d}{b}} \leftarrow$ partition $\pi$ into $\frac{d}{b}$ pieces.
$W \leftarrow 0_{d \times k}$.
$R \leftarrow 0_{n \times k}$
for $\ell = 1$ to $n_e$ do
  $\pi \leftarrow$ random permutation of $\{1, ..., \frac{d}{b}\}$.
  for $i = 1$ to $\frac{d}{b}$ do
    $X_b \leftarrow$ FEATUREBLOCK($X, \mathcal{I}_{\pi_i}$). /* Column block. */
    $R \leftarrow R - X_b W(\mathcal{I}_{\pi_i}, [k])$.
    Solve $(X_b^\mathsf{T} X_b + n\lambda I_b) W_b = X_b^\mathsf{T} (Y - R)$.
    $R \leftarrow R + X_b W_b$.
    $W(\mathcal{I}_{\pi_i}, [k]) \leftarrow W_b$.
```

at each iteration b rows of the feature matrix are sampled. This smaller matrix (X_b) is then used to compute the gradient and the final value of the gradient is used to update the model taking into account the step size s.

Block Coordinate Descent: To see how column sampling is used, we present a block-coordinate descent (BCD) algorithm in Algorithm 2. The BCD algorithm works by sampling a block b of coordinates (or columns) from the feature matrix at each iteration. For quadratic loss, this column block X_b is then used to compute an update. We only update values for the selected b coordinates and keep the other coordinates constant. This process is repeated on every iteration. A common sampling scheme is to split the coordinates into blocks within an epoch and run a specified number of epochs.

Having discussed algorithms that use row and column sampling we next turn our attention to real-world end-to-end machine learning applications. We present two examples again: one where we deal with a sparse feature matrix and another with a dense feature matrix. These examples are drawn from the KeystoneML [157] project.

Text Analytics. Text classification problems typically involve tasks which start with raw data from various sources e.g. from newsgroups, emails or a Wikipedia dataset. Common text classification pipelines include featurization steps like pre-processing the raw data to form N-grams, filtering of stop words, part-of-speech tagging or named entity recognition. Existing packages like CoreNLP [118] perform featurization for small scale datasets on a single machine. After performing featurization, developers typically learn a model using NaiveBayes or SVM-based classifiers. Note that the data here usually has a large number of features and is very *sparse*. As an example, consider a pipeline to classify product reviews from the Amazon Reviews dataset [120]. The

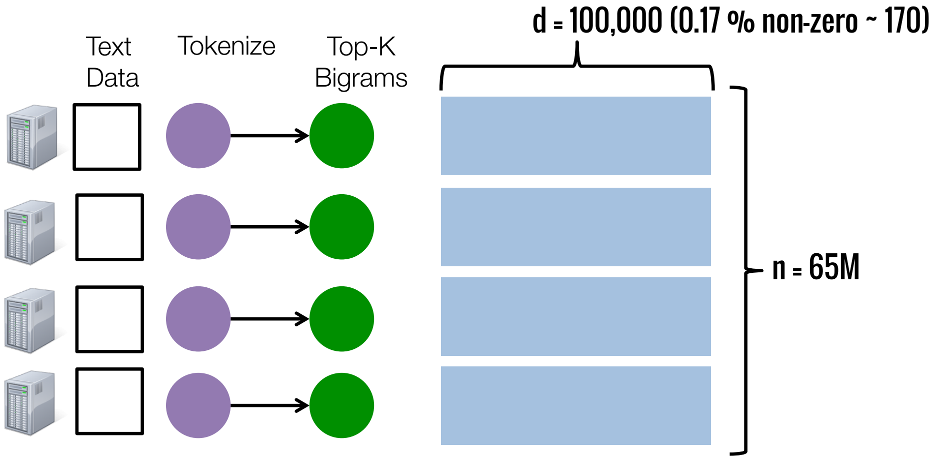


Figure 2.1: Execution of a machine learning pipeline used for text analytics. The pipeline consists of featurization and model building steps which are repeated for many iterations.

dataset is a collection of approximately 65 million product reviews, rated from 1 to 5 stars. We can build a classifier to predict polarity of a review by chaining together nodes as shown in Figure 2.1. The first step of this pipeline is tokenization, followed by an TopK bigram operator which extracts most common bigrams from the document. We finally build a model using a Logistic Regression with mini-batch SGD.

Speech Recognition Pipeline. As another example we consider a speech recognition pipeline [90] that achieves state-of-the-art accuracy on the TIMIT [35] dataset. The pipeline trains a model using kernel SVMs and the execution DAG is shown in Figure 2.2. From the figure we can see that pipeline contains three main stages. The first stage reads input data, and featurizes the data by applying MFCC [190]. Following that it applies a random cosine transformation [141] to each record which generates a dense feature matrix. In the last stage, the features are fed into a block coordinate descent based solver to build a model. The model is then refined by generating more features and these steps are repeated for 100 iterations to achieve state-of-the-art accuracy.

In summary, in this section we surveyed the main characteristics of machine learning workloads and showed how the properties of approximation, sampling are found in iterative solvers. Finally using real world example we also studied how data density could vary across applications. We next look at how these applications are executed on a distributed system to derive systems characteristics that can be optimized to improve performance.

2.2 Execution Phases

To understand the system characteristics that influence the performance of machine learning algorithms we first study how the two algorithms presented in the previous section can be executed

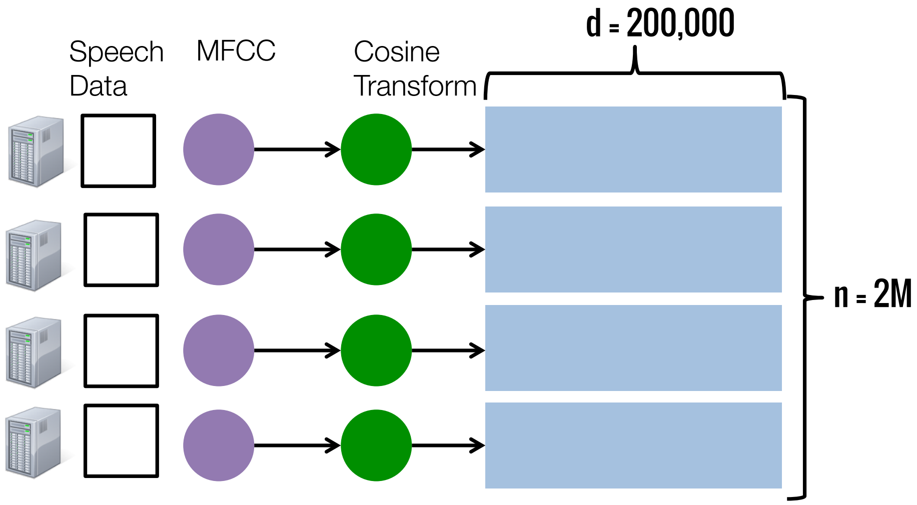


Figure 2.2: Execution DAG of a machine learning pipeline used for speech recognition. The pipeline consists of featurization and model building steps which are repeated for many iterations.

using a distributed architecture. We study distributed implementations of mini-batch SGD [59] and block coordinate descent [162] designed for the message passing computing model [24]. The message passing model consists of a number of independent machines connected by communication network. When compared to the shared memory model, message passing is particularly beneficial while modeling scenarios where the communication latency between machines is high.

Figure 2.3 shows how the two algorithms are implemented in a message passing model. We begin by assuming that the training data is partitioned across the machines in the cluster. In the case of mini-batch SGD we compute a sample of size b and correspondingly can launch computation on the machines which have access to the sampled data. In each of these computation *tasks*, we calculate the gradient for the data points in that partition. The results from these tasks are aggregated to compute the final gradient.

In the case of block coordinate descent, a column block is partitioned across the cluster of machines. Similar to SGD, we launch computation on machines in the form of tasks and in this case the tasks compute $X_i^T X_i$ and $X_i^T Y_i$. The results are again aggregated to get the final values that can then be plugged into the update rule shown in Algorithm 2.

From the above description we can see that both the executions have very similar phases from a systems perspective. Broadly we can define each iteration to do the following steps. First the necessary input data is *read* from storage (row or column samples) and then *computation* is performed on this data. Following that the results from all the tasks are *aggregated*. Finally the model update is computed and this captures one iteration of execution. To run the next iteration the update model value from the previous iteration is typically required and hence this updated model is *broadcasted* to all the machines.

These four phases of read, compute, aggregate and broadcast capture the execution workflow for a diverse set of machine learning algorithms. This characterization is important from a systems perspective as instead of making improvements to specific algorithms we can instead develop

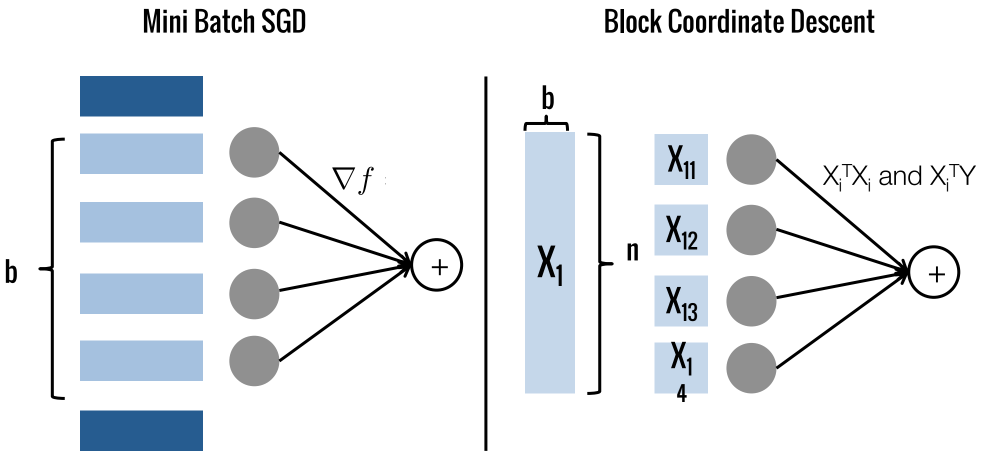


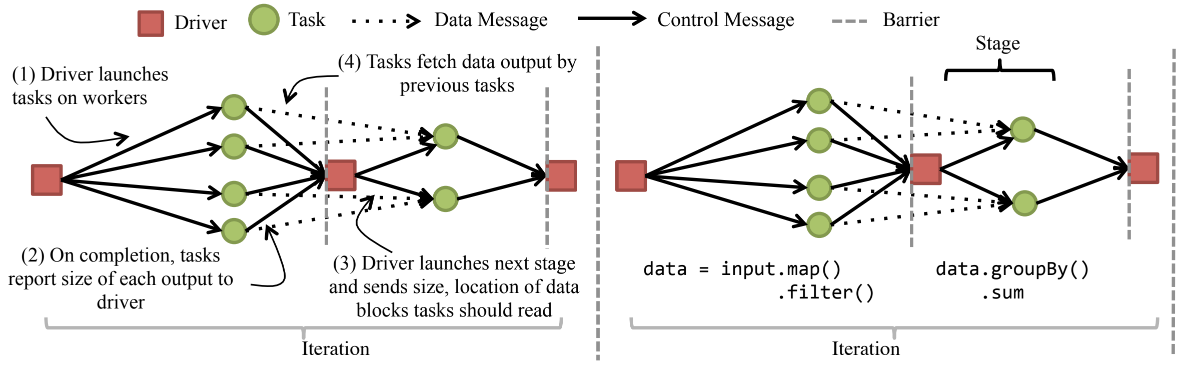
Figure 2.3: Execution of Mini-batch SGD and Block coordinate descent on a distributed runtime.

more general solutions to accelerate each of these phases. We next look at how these phases are implemented in distributed data processing frameworks.

2.3 Computation Model

One of the more popular computation models used by a number of recent distributed data processing frameworks is the bulk-synchronous parallel (BSP) model [166]. In this model, the computation consists of a phase whereby all parallel nodes in the system perform some local computation, followed by a blocking *barrier* that enables all nodes to communicate with each other, after which the process repeats itself. The MapReduce [57] paradigm adheres to this model, whereby a *map* phase can do arbitrary local computations, followed by a barrier in the form of an all-to-all shuffle, after which the *reduce* phase can proceed with each reducer reading the output of relevant mappers (often all of them). Systems such as Dryad [91, 182], Spark [184], and FlumeJava [39] extend the MapReduce model to allow combining many phases of map and reduce after each other, and also include specialized operators, e.g. filter, sum, group-by, join. Thus, the computation is a directed acyclic graph (DAG) of operators and is partitioned into different *stages* with a barrier between each of them. Within each stage, many map functions can be fused together as shown in Figure 2.4. Further, many operators (e.g., sum, reduce) can be efficiently implemented [20] by pre-combining data in the map stage and thus reducing the amount of data transferred.

Coordination at barriers greatly simplifies fault-tolerance and scaling in BSP systems. First, the scheduler is notified at the end of each stage, and can reschedule tasks as necessary. This in particular means that the scheduler can add parallelism at the end of each stage, and use additional machines when launching tasks for the next stage. Furthermore, fault tolerance in these systems is typically implemented by taking a consistent snapshot at each barrier. This snapshot can either be physical, *i.e.,* record the output from each task in a stage; or logical, *i.e.,* record the computational


Driver
Task
Data Message
Control Message
Barrier
Stage
(1) Driver launches tasks on workers
(4) Tasks fetch data output by previous tasks
(2) On completion, tasks report size of each output to driver
(3) Driver launches next stage and sends size, location of data blocks tasks should read
Iteration
Iteration

```
data = input.map()
            .filter()
```

```
data.groupBy()
    .sum
```

Figure 2.4: Execution of a job when using the batch processing model. We show two iterations of execution here. The left-hand side shows the various steps used to coordinate execution. The query being executed in shown on the right hand side

dependencies for some data. Task failures can be trivially handled using these snapshots since the scheduler can reschedule the task and have it read (or reconstruct) inputs from the snapshot.

However the presence of barriers results in performance limits when using the BSP model. If we denote the time per iteration as T, then T cannot be set to adequately small values due to how barriers are implemented in these systems. Consider a simple job consisting of a map phase followed by a reduce phase (Figure 2.4). A centralized driver schedules all the map tasks to take turns running on free resources in the cluster. Each map task then outputs records for each reducer based on some partition strategy, such as hashing or sorting. Each task then informs the centralized driver of the allocation of output records to the different reducers. The driver can then schedule the reduce tasks on available cluster resources, and pass this metadata to each reduce task, which then fetches the relevant records from all the different map outputs. Thus, each barrier in a iteration requires communicating back and forth with the driver. Hence, if we aim for T to be too low this will result in a substantial driver communication and scheduling overhead, whereby the communication with the driver eventually dominates the processing time. In most systems, T is limited to 0.5 seconds or more [179].

2.4 Related Work

We next survey some of the related research in the area of designing systems for large scale machine learning. We describe other efforts to improve performance for data analytics workloads and other system designs used for low latency execution. We also discuss performance modeling and performance prediction techniques from prior work in systems and databases.

2.4.1 Cluster scheduling

Cluster scheduling has been an area of active research and recent work has proposed techniques to enforce fairness [70, 93], satisfy job constraints [71] and improve locality [93, 183]. Straggler mitigation solutions launch extra copies of tasks to mitigate the impact of slow running tasks [13, 15, 178, 186]. Further, systems like Borg [173], YARN [17] and Mesos [88] schedule jobs from different frameworks on a shared cluster. Prior work [135] has also identified the benefits of shorter task durations and this has led to the development of distributed job schedulers such as Sparrow [136], Apollo [30], etc. These scheduling frameworks focus on scheduling across jobs while we study scheduling within a single machine learning job. To improve performance within a job, techniques for improving data locality [14, 184], re-optimizing queries [103], dynamically distributing tasks [130] and accelerating network transfers [49, 78] have been proposed. Prior work [172] has also looked at the benefits of removing the barrier across shuffle stages to improve performance. In this thesis we focus on machine learning jobs and how can exploit the specific properties they have to get better performance.

2.4.2 Machine learning frameworks

Recently, a large body of work has focused on building cluster computing frameworks that support machine learning tasks. Examples include GraphLab [72, 117], Spark [184], DistBelief [56], Tensorflow [3], Caffe [97], MLBase [105] and KeystoneML [157]. Of these, GraphLab and Spark add support for abstractions commonly used in machine learning. Neither of these frameworks provide any explicit system support for sampling. For instance, while Spark provides a sampling operator, this operation is carried out entirely in application logic, and the Spark scheduler is oblivious to the use of sampling. Further the BSP model in Spark introduces scheduling overheads as discussed in Section 2.3. MLBase and KeystoneML present a declarative programming model to simplify constructing machine learning applications. Our focus in this thesis is on how we can accelerate the performance of the underlying execution engine and we seek to build systems that are compatible with the APIs from KeystoneML. Finally, while Tensorflow, Caffe and DistBelief are tuned to running large deep learning workloads [107], we focus on general design techniques that can apply to a number of algorithms like SGD, BCD etc.

2.4.3 Continuous Operator Systems

While we highlighted BSP-style frameworks in Section 2.3, an alternate computation model that is used is the dataflow [98] computation model with long running or *continuous operators*. Dataflow models have been used to build database systems [73], streaming databases [2, 40] and have been extended to support distributed execution in systems like Naiad [129], StreamScope [116] and Flink [144]. In such systems, similar to BSP frameworks, user programs are converted to a DAG of operators, and each operator is placed on a processor as a long running task. A processor may contain a number of long running tasks. As data is processed, operators update local state and messages are directly transferred from between operators. Barriers are inserted only when required by specific operators. Thus, unlike BSP-based systems, there is no scheduling or communication overhead with a centralized driver. Unlike BSP-based systems, which require a barrier at the end of a micro-batch, continuous operator systems do not impose any such barriers.

To handle machine failures, continuous operator systems typically use distributed checkpointing algorithms [41] to create consistent snapshots periodically. The execution model is flexible and can accommodate either asynchronous [36] checkpoints (in systems like Flink) or synchronous checkpoints (in systems like Naiad). Recent work [92] provides a more detailed description comparing these two approaches and also describes how the amount of state that is checkpointed can be minimized. However checkpoint replay during recovery can be more expensive in this model. In both synchronous and asynchronous approaches, whenever a node fails, all the nodes are rolled back to the last consistent checkpoint and records are then replayed from this point. As the continuous operators cannot be easily split into smaller components this precludes parallelizing recovery across timesteps (as in the BSP model) and each continuous operator is recovered serially. In this thesis we focus on re-using existing fault tolerance semantics from BSP systems and improving performance for machine learning workloads.

2.4.4 Performance Prediction

There have been a number of recent efforts at modeling job performance in datacenters to support SLOs or deadlines. Techniques proposed in Jockey [66] and ARIA [171] use historical traces and dynamically adjust resource allocations in order to meet deadlines. Bazaar [95] proposed techniques to model the network utilization of MapReduce jobs by using small subsets of data. Projects like MRTuner [149] and Starfish [87] model MapReduce jobs at very fine granularity and set optimal values for options like memory buffer sizes etc. Finally scheduling frameworks like Quasar [60] try to estimate the scale out and scale up factor for jobs using the progress rate of the first few tasks. In this thesis our focus is on modeling machine learning workloads and being able to minimize the amount of time spent in developing such a model. In addition we aim to extract performance characteristics that are not specific to MapReduce implementations and are independent of the framework, number of stages of execution etc.

2.4.5 Database Query Optimization

Database query progress predictors [44, 127] also solve a performance prediction problem. Database systems typically use summary statistics [146] of the data like cardinality counts to guide this process. Further, these techniques are typically applied to a known set of relational operators. Similar ideas have also been applied to linear algebra operators [89]. In this thesis we aim to handle a large class of machine learning jobs where we only know high level properties of the computation being run. Recent work has also looked at providing SLAs for OLTP [134] and OLAP workloads [85] in the cloud and some of our motivation about modeling cloud computing instances are also applicable to database queries.

2.4.6 Performance optimization, Tuning

Recent work including Nimbus [119] and Thrill [23] has focused on implementing high-performance BSP systems. Both systems claim that the choice of runtime (*i.e.,* JVM) has a major effect on performance, and choose to implement their execution engines in C++. Furthermore, Nimbus similar to our work finds that the scheduler is a bottleneck for iterative jobs and uses scheduling templates. However, during execution Nimbus uses mutable state and focuses on HPC applications while we focus on improving adaptivity for machine learning workloads. On the other hand Thrill focuses on query optimization in the data plane.

Ideas related to our approach to deployment, where we explore a space of possible configurations and choose the best configuration, have been used in other applications like server benchmarking [150]. Related techniques like Latin Hypercube Sampling have also been used to efficiently explore file system design space [84]. Auto-tuning BLAS libraries [22] like ATLAS [51] also solve a similar problem of exploring a state space efficiently to prescribe the best configuration.

Chapter 3

Modeling Machine Learning Jobs

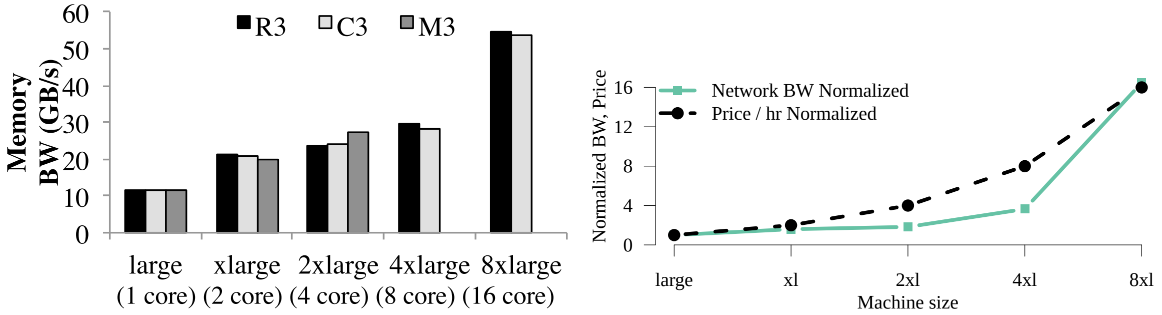
Having looked at the main properties of machine learning workloads in Chapter 2, in this chapter we study the problem of developing performance models for distributed machine learning jobs. Using performance models we aim to understand how the running time changes as we modify the input size and the cluster size used to run the workload. Our key contribution in this chapter is to exploit the workload properties (§1.1) i.e., *approximation*, *iteration*, *sampling* and *computational structure* to make it cheaper to build performance models.

Performance models are also useful in a cloud computing setting for choosing the right hardware configuration. The choice of configuration depends on the user's goals which typically includes either minimizing the running time given a budget or meeting a deadline while minimizing the cost. One way to address this problem is developing a performance prediction framework that can accurately predict the running time on a specified hardware configuration, given a job and its input.

We propose, Ernest, a performance prediction framework that can provide accurate predictions with low overhead. The main idea in Ernest is to run a set of instances of the machine learning job on samples of the input, and use the data from these runs to create a performance model. This approach has low overhead, as in general it takes much less time and resources to run the job on samples than running the entire job. The reason this approach works is that many machine learning workloads have a simple structure and the dependence between their running times and the input sizes or number of nodes is in general characterized by a relatively small number of smooth functions.

The cost and utility of training data points collected is important for low-overhead prediction and we address this problem using *optimal experiment design* [139], a statistical technique that allows us to select the most useful data points for training. We augment experiment design with a cost model and this helps us find the training data points to explore within a given budget.

As our methods are also applicable to other workloads like graph processing and scientific workloads in genomics, we collectively address these workloads as *advanced analytics workloads*. We evaluate Ernest using a number of workloads including (a) several machine learning algorithms that are part of Spark MLlib [124], (b) queries from GenBase [160] and I/O intensive transformations using ADAM [133] on a full genome, and (c) a speech recognition pipeline that achieves



(a) Comparison of memory bandwidths across Amazon EC2 `m3`/`c3`/`r3` instance types. There are only three sizes for `m3`. Smaller instances (large, xlarge) have better memory bandwidth per core.

(b) Comparison of network bandwidths with prices across different EC2 `r3` instance sizes normalized to `r3.large`. `r3.8xlarge` has the highest bandwidth per core.

Figure 3.1: Memory bandwidth and network bandwidth comparison across instance types

state-of-the-art results [90]. Our evaluation shows that our average prediction error is under 20% and that this is sufficient for choosing the appropriate number or type of instances. Our training overhead for long-running jobs is less than 5% and we also find that using experiment design improves prediction error for some algorithms by $30-50\%$ over a cost-based scheme

3.1 Performance Prediction Background

We first present an overview of different approaches to performance prediction. We then discuss recent hardware trends in computation clusters that make this problem important and finally discuss some of the computation and communication patterns that we see in machine learning workloads.

3.1.1 Performance Prediction

Performance modeling and prediction have been used in many different contexts in various systems [21, 66, 131]. At a high level performance modeling and prediction proceeds as follows: select an output or response variable that needs to be predicted and the features to be used for prediction. Next, choose a relationship or a model that can provide a prediction for the output variable given the input features. This model could be rule based [25, 38] or use machine learning techniques [132, 178] that build an estimator using some training data. We focus on machine learning based techniques in this chapter and we next discuss two major approaches in modeling that influences the training data and machine learning algorithms used.

Performance counters: Performance counter based approaches typically use a large number of low level counters to try and predict application performance characteristics. Such an approach has been used with CPU counter for profiling [16], performance diagnosis [33, 180] and virtual machine allocation [132]. A similar approach has also been used for analytics jobs where the MapReduce counters have been used for performance prediction [171] and straggler mitigation [178]. Performance-counter based approaches typically use advanced learning algorithms like random forests, SVMs. However as they use a large number of features, they require large amounts of training data and are well suited for scenarios where historical data is available.

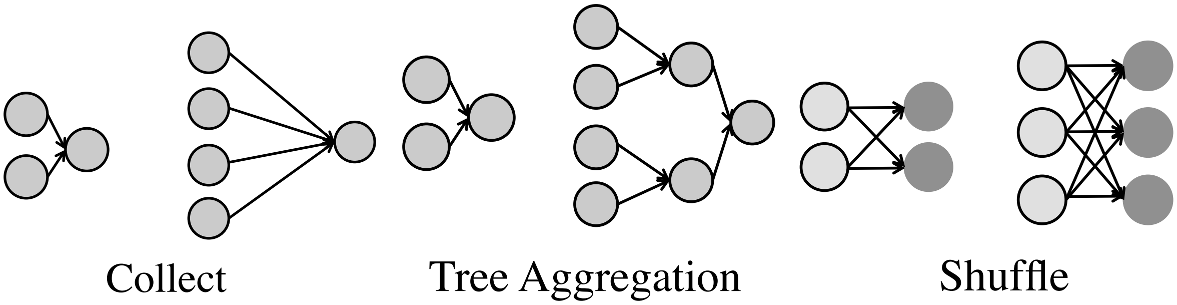
System modeling: In the system modeling approach, a performance model is developed based on the properties of the system being studied. This method has been used in scientific computing [21] for compilers [5], programming models [25, 38]; and by databases [44, 127] for estimating the progress made by SQL queries. System design based models are usually simple and interpretable but may not capture all the execution scenarios. However one advantage of this approach is that only a small amount of training data is required to make predictions.

In this chapter, we look at how to perform efficient performance prediction for large scale advanced analytics. We use a system modeling approach where we build a high-level end-to-end model for advanced analytics jobs. As collecting training data can be expensive, we further focus on how to minimize the amount of training data required in this setting. We next survey recent hardware and workload trends that motivate this problem.

3.1.2 Hardware Trends

The widespread adoption of cloud computing has led to a large number of data analysis jobs being run on cloud computing platforms like Amazon EC2, Microsoft Azure and Google Compute Engine. In fact, a recent survey by Typesafe of around 500 enterprises [164] shows that 53% of Apache Spark users deploy their code on Amazon EC2. However using cloud computing instances comes with its own set of challenges. As cloud computing providers use virtual machines for isolation between users, there are a number of fixed-size virtual machine options that users can choose from. Instance types vary not only in capacity (i.e. memory size, number of cores etc.) but also in performance. For example, we measured memory bandwidth and network bandwidth across a number of instance types on Amazon EC2. From Figure 3.1(a) we can see that the smaller instances i.e. `large` or `xlarge` have the highest memory bandwidth available per core while Figure 3.1(b) shows that `8xlarge` instances have the highest network bandwidth available per core. Based on our experiences with Amazon EC2, we believe these performance variations are not necessarily due to poor isolation between tenants but are instead related to how various instance types are mapped to shared physical hardware.

The non-linear relationship between price vs. performance is not only reflected in microbenchmarks but can also have a significant effect on end-to-end performance. For example, we use two machine learning kernels: (a) A least squares solver used in convex optimization [61] and (b) a matrix multiply operation [167], and measure their performance for similar capacity configurations across a number of instance types. The results (Figure 3.3(a)) show that picking the right instance type can improve performance by up to 1.9x at the same cost for the least squares solver. Earlier studies [86, 175] have also reported such performance variations for other applications like SQL queries, key-value stores. These performance variations motivate the need for a performance prediction framework that can automate the choice of hardware for a given computation.



(a) (b) (c)

Figure 3.2: Scaling behaviors of commonly found communication patterns as we increase the number of machines.

Finally, performance prediction is important not just in cloud computing but it is also useful in other shared computing scenarios like private clusters. Cluster schedulers [17] typically try to maximize utilization by packing many jobs on a single machine and predicting the amount of memory or number of CPU cores required for a computation can improve utilization [60]. Next, we look at workload trends in large scale data analysis and how we can exploit workload characteristics for performance prediction.

Workload Properties: As discussed in Chapter 2, the last few years have seen the growth of advanced analytics workloads like machine learning, graph processing and scientific analyses on large datasets. Advanced analytics workloads are commonly implemented on top of data processing frameworks like Hadoop [57], Naiad [129] or Spark [184] and a number of high level libraries for machine learning [18, 124] have been developed on top of these frameworks. A survey [164] of Apache Spark users shows that around 59% of them use the machine learning library in Spark and recently launched services like Azure ML [125] provide high level APIs which implement commonly used machine learning algorithms.

Advanced analytics workloads differ from other workloads like SQL queries or stream processing in a number of ways (Section 1.1). These workloads are typically numerically intensive, i.e. performing floating point operations like matrix-vector multiplication or convolutions [52], and thus are sensitive to the number of cores and memory bandwidth available. Further, such workloads are also often iterative and repeatedly perform parallel operations on data cached in memory across a cluster. Advanced analytics jobs can also be long-running: for example, to obtain the state-of-the-art accuracy on tasks like image recognition [56] and speech recognition [90], jobs are run for many hours or days.

Since advanced analytics jobs run on large datasets are expensive, we observe that developers have focused on algorithms that are scalable across machines and are of low complexity (e.g., linear or quasi-linear) [29]. Otherwise, using these algorithms to process huge amounts of data might be infeasible. The natural outcome of these efforts is that these workloads admit relatively simple performance models. Specifically, we find that the computation required per data item remains the same as we scale the computation.

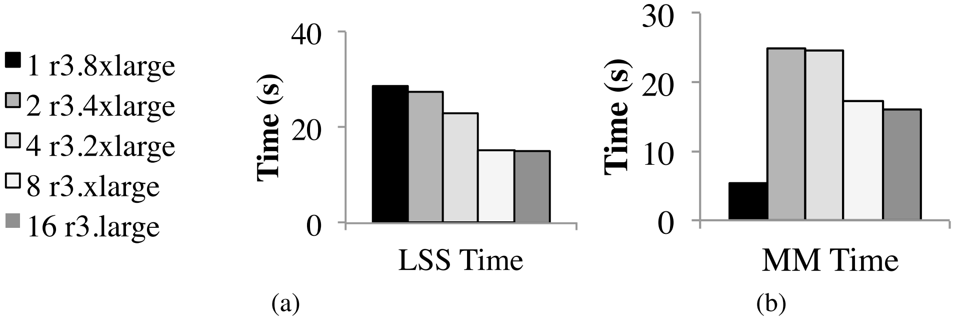


Figure 3.3: Performance comparison of a Least Squares Solver (LSS) job and Matrix Multiply (MM) across similar capacity configurations.

Further, we observe that only a few communication patterns repeatedly occur in such jobs. These patterns (Figure 3.2) include (a) the all-to-one or *collect* pattern, where data from all the partitions is sent to one machine, (b) *tree-aggregation* pattern where data is aggregated using a tree-like structure, and (c) a *shuffle* pattern where data goes from many source machines to many destinations. These patterns are not specific to advanced analytics jobs and have been studied before [24,48]. Having a handful of such patterns means that we can try to automatically infer how the communication costs change as we increase the scale of computation. For example, assuming that data grows as we add more machines (i.e., the data per machine is constant), the time taken for the collect increases as $O(machines)$ as a single machine needs to receive all the data. Similarly the time taken for a binary aggregation tree grows as $O(log(machines))$.

Finally we observe that many algorithms are iterative in nature and that we can also *sample the computation* by running just a few iterations of the algorithm. Next we will look at the design of the performance model.

3.2 Ernest Design

In this section we outline a model for predicting execution time of advanced analytics jobs. This scheme only uses end-to-end running times collected from executing the job on smaller samples of the input and we discuss techniques for model building and data collection.

At a high level we consider a scenario where a user provides as input a parallel job (written using any existing data processing framework) and a pointer to the input data for the job. We do not assume the presence of any historical logs about the job and our goal here is to build a model that will predict the execution time for any input size, number of machines for this given job. The main steps in building a predictive model are (a) determining what training data points to collect (b) determining what features should be derived from the training data and (c) performing feature selection to pick the simplest model that best fits the data. We discuss all three aspects below.

3.2.1 Features for Prediction

One of the consequences of modeling end-to-end unmodified jobs is that there are only a few parameters that we can change to observe changes in performance. Assuming that the job, the dataset and the machine types are fixed, the two main features that we have are (a) the number of rows or fraction of data used (scale) and (b) the number of machines used for execution. Our goal in the modeling process is to derive as few features as required for the amount of training data required grows linearly with the number of features.

To build our model we add terms related to the computation and communication patterns discussed in §2.1. The terms we add to our linear model are (a) a fixed cost term which represents the amount of time spent in serial computation (b) the interaction between the scale and the inverse of the number of machines; this is to capture the *parallel* computation time for algorithms whose computation scales *linearly* with data, i.e., if we double the size of the data with the same number of machines, the computation time will grow linearly (c) a $log(machines)$ term to model communication patterns like aggregation trees (d) a linear term $O(machines)$ which captures the all-to-one communication pattern and fixed overheads like scheduling / serializing tasks (i.e. overheads that scale as we add more machines to the system). Note that as we use a linear combination of *non-linear features*, we can model non-linear behavior as well.

Thus the overall model we are fitting tries to learn values for θ_0, θ_1, θ_2, and θ_3 in the formula

$$\begin{aligned} time = \theta_0 + \theta_1 \times (scale \times \frac{1}{machines}) + \\ \theta_2 \times \log(machines) + \\ \theta_3 \times machines \end{aligned} \tag{3.1}$$

Given these features, we then use a *non-negative least squares* (NNLS) solver to find the model that best fits the training data. NNLS fits our use case very well as it ensures that each term contributes some non-negative amount to the overall time taken. This avoids over-fitting and also avoids corner cases where say the running time could become negative as we increase the number of machines. NNLS is also useful for feature selection as it sets coefficients which are not relevant to a particular job to zero. For example, we trained a NNLS model using 7 data points on all of the machine learning algorithms that are a part of MLlib in Apache Spark 1.2. The final model parameters are shown in Table 3.1. From the table we can see two main characteristics: (a) that not all features are used by every algorithm and (b) that the contribution of each term differs for each algorithm. These results also show why we cannot reuse models across jobs.

Additional Features: While the features used above capture most of the patterns that we see in jobs, there could other patterns which are not covered. For example in linear algebra operators like QR decomposition the computation time will grow as ${}^{scale^2}/_{machines}$ if we scale the number of columns. We discuss techniques to detect when the model needs such additional terms in §3.2.4.

Benchmark	$intercept$	$^{scale}/_{mc}$	mc	$log(mc)$
spearman	0.00	4887.10	0.00	4.14
classification	0.80	211.18	0.01	0.90
pca	6.86	208.44	0.02	0.00
naive.bayes	0.00	307.48	0.00	1.00
summary stats	0.42	39.02	0.00	0.07
regression	0.64	630.93	0.09	1.50
als	28.62	3361.89	0.00	0.00
kmeans	0.00	149.58	0.05	0.54

Table 3.1: Models built by Non-Negative Least Squares for MLlib algorithms using `r3.xlarge` instances. Not all features are used by every algorithm.

3.2.2 Data collection

The next step is to collect training data points for building a predictive model. For this we use the input data provided by the user and run the *complete* job on small samples of the data and collect the time taken for the job to execute. For iterative jobs we allow Ernest to be configured to run a certain number of iterations (§3.3). As we are not concerned with the accuracy of the computation we just use the first few rows of the input data to get appropriately sized inputs.

How much training data do we need?: One of the main challenges in predictive modeling is minimizing the time spent on collecting training data while achieving good enough accuracy. As with most machine learning tasks, collecting more data points will help us build a better model but there is time and a cost associated with collecting training data. As an example, consider the model shown in Table 3.1 for *kmeans*. To train this model we used 7 data points and we look at the importance of collecting additional data by comparing two schemes: in the first scheme we collect data in an increasing order of machines and in the second scheme we use a mixed strategy as shown in Figure 3.4. From the figure we make two important observations: (a) in this case, the mixed strategy gets to a lower error quickly; after three data points we get to less than 15% error. (b) We see a trend of diminishing returns where adding more data points does not improve accuracy by much. We next look at techniques that will help us find how much training data is required and what those data points should be.

3.2.3 Optimal Experiment Design

To improve the time taken for training without sacrificing the prediction accuracy, we outline a scheme based on *optimal experiment design*, a statistical technique that can be used to minimize the number of experiment runs required. In statistics, experiment design [139] refers to the study of how to collect data required for any experiment given the modeling task at hand. *Optimal* experiment design specifically looks at how to choose experiments that are optimal with respect to some statistical criterion. At a high-level the goal of experiment design is to determine data points that can give us most information to build an accurate model. some subset of training data points and then determine how far a model trained with those data points is from the ideal model.

More formally, consider a problem where we are trying to fit a linear model X given measurements $y_1, \ldots, y_m$ and features $a_1, \ldots, a_m$ for each measurement. Each feature vector could in turn consist of a number of dimensions (say n dimensions). In the case of a linear model we typically estimate X using linear regression. We denote this estimate as $\hat{X}$ and $\hat{X} - X$ is the estimation error or a measure of how far our model is from the true model.

To measure estimation error we can compute the Mean Squared Error (MSE) which takes into account both the bias and the variance of the estimator. In the case of the linear model above if we have m data points each having n features, then the variance of the estimator is represented by the $n \times n$ covariance matrix $(\sum\limits_{i=1}^{m} a_i a_i^T)^{-1}$. The key point to note here is that the covariance matrix only depends on the feature vectors that were used for this experiment and not on the model that we are estimating.

In optimal experiment design we choose feature vectors (i.e. a_i) that minimize the estimation error. Thus we can frame this as an optimization problem where we minimize the estimation error subject to constraints on the number of experiments. More formally we can set λ_i as the fraction of times an experiment is chosen and minimize the trace of the inverse of the covariance matrix:

$$\begin{array}{ll} \text{Minimize} & \mathbf{tr}((\sum\limits_{i=1}^{m} \lambda_i a_i a_i^T)^{-1}) \\ \text{subject to} & \lambda_i \geq 0, \lambda_i \leq 1 \end{array}$$

Using Experiment Design: The predictive model described in the previous section can be formulated as an experiment design problem. Given bounds for the scale and number of machines we want to explore, we can come up with all the features that can be used. For example if the scale bounds range from say 1% to 10% of the data and the number of machine we can use ranges from 1 to 5, we can enumerate 50 different feature vectors from all the scale and machine values possible. We can then feed these feature vectors into the experiment design setup described above and only choose to run those experiments whose λ values are non-zero.

Accounting for Cost: One additional factor we need to consider in using experiment design is that each experiment we run costs a different amount. This cost could be in terms of time (i.e. it is more expensive to train with larger fraction of the input) or in terms of machines (i.e. there is a fixed cost to say launching a machine). To account for the cost of an experiment we can augment the optimization problem we setup above with an additional constraint that the total cost should be lesser than some budget. That is if we have a cost function which gives us a cost c_i for an experiment with scale s_i and m_i machines, we add a constraint to our solver that $\sum\limits_{i=1}^{m} c_i \lambda_i \leq B$ where B is the total budget. For the rest of this chapter we use the time taken to collect training data as the cost and ignore any machine setup costs as we usually amortize that over all the data we need to collect. However we can plug-in in any user-defined cost function in our framework.

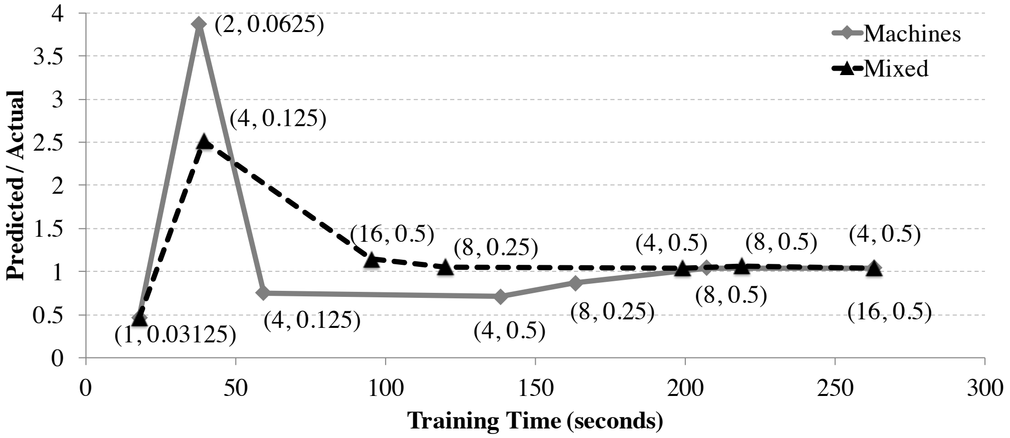


Figure 3.4: Comparison of different strategies used to collect training data points for KMeans. The labels next to the data points show the (number of machines, scale factor) used.

	Residual Sum	Percentage Err	
	of Squares	Median	Max
without $\sqrt{n}$	1409.11	12.2%	64.9%
with $\sqrt{n}$	463.32	5.7%	26.5%

Table 3.2: Cross validation metrics comparing different models for Sparse GLM run on the splice-site dataset.

3.2.4 Model extensions

The model outlined in the previous section accounts for the most common patterns we see in advanced analytics applications. However there are some complex applications like randomized linear algebra [82] which might not fit this model. For such scenarios we discuss two steps: the first is adding support in Ernest to detect when the model is not adequate and the second is to easily allow users to extend the model being used.

Cross-Validation: The most common technique for testing if a model is valid is to use hypothesis testing and compute test statistics (e.g., using the t-test or the chi-squared test) and confirm the null hypothesis that data belongs to the distribution that the model describes. However as we use non-negative least squares (NNLS) the residual errors are not normally distributed and simple techniques for computing confidence limits, p-values are not applicable. Thus we use *cross-validation*, where subsets of the training data can be used to check if the model will generalize well. There are a number of methods to do cross-validation and as our training data size is small, we use a leave-one-out-cross-validation scheme in Ernest. Specifically if we have collected m training data points, we perform m cross-validation runs where each run uses $m - 1$ points as training data and tests the model on the left out data point. across the runs.

Model extension example: As an example, we consider the GLM classification implementation in Spark MLLib for sparse datasets. In this workload the computation is linear but the aggregation uses two stages (instead of an aggregation tree) where the first aggregation stage has $\sqrt{n}$ tasks for n partitions of data and the second aggregation stage combines the output of $\sqrt{n}$ tasks using one task. This communication pattern is not captured in our model from earlier and the results from cross validation using our original model are shown in Table 3.2. As we can see in the table both the residual sum of squares and the percentage error in prediction are high for the original model. Extending the model in Ernest with additional terms is simple and in this case we can see that adding the $\sqrt{n}$ term makes the model fit much better. In practice we use a configurable threshold on the percentage error to determine if the model fit is poor. We investigate the end-to-end effects of using a better model in §3.5.6.

3.3 Ernest Implementation

Ernest is implemented using Python as multiple modules. The modules include a job submission tool that submits training jobs, a training data selection process which implements experiment design using a CVX solver [74, 75] and finally a model builder that uses NNLS from SciPy [100]. Even for a large range of scale and machine values we find that building a model takes only a few seconds and does not add any overhead. In the rest of this section we discuss the job submission tool and how we handle sparse datasets, stragglers.

3.3.1 Job Submission Tool

Ernest extends existing job submission API [155] that is present in Apache Spark 1.2. This job submission API is similar to Hadoop's Job API [80] and similar job submission APIs exist for dedicated clusters [142, 173] as well. The job submission API already takes in the binary that needs to run (a JAR file in the case of Spark) and the input specification required for collecting training data.

We add a number of optional parameters which can be used to configure Ernest. Users can configure the minimum and maximum dataset size that will be used for training. Similarly the maximum number of machines to be used for training can also be configured. Our prototype implementation of Ernest uses Amazon EC2 and we amortize cluster launch overheads across multiple training runs i.e., if we want to train using 1, 2, 4 and 8 machines, we launch a 8 machine cluster and then run all of these training jobs in parallel.

The model built using Ernest can be used in a number of ways. In this chapter we focus on a cloud computing use case where we can choose the number and type of EC2 instances to use for a given application. To do this we build one model per instance type and explore different sized instances (i.e. r3.large,...r3.8xlarge). After training the models we can answer higher level questions like selecting the cheapest configuration given a time bound or picking the fastest configuration given a budget. One of the challenges in translating the performance prediction into a higher-level decision is that the predictions could have some error associated with them. To help with this, we

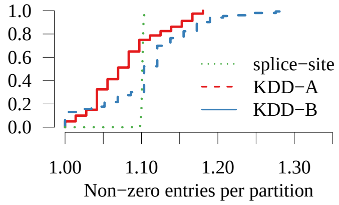


Figure 3.5: CDF of maximum number of non-zero entries in a partition, normalized to the least loaded partition for sparse datasets.

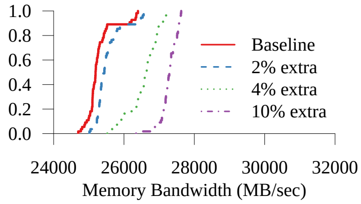


Figure 3.6: CDFs of STREAM memory bandwidths under four allocation strategies. Using a small percentage of extra instances removes stragglers.

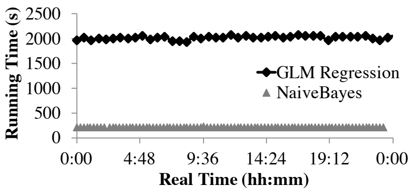


Figure 3.7: Running times of GLM and Naive Bayes over a 24-hour time window on a 64-node EC2 cluster.

provide the cross validation results (§3.2.4) along with the prediction and these can be used to compute the range of errors observed on training data. Additionally we plan to provide support for visualizing the scaling behavior and Figure 3.14(b) in §3.5.6 shows an example.

3.3.2 Handling Sparse Datasets

One of the challenges in Ernest is to deal with algorithms that process sparse datasets. Because of the difference in sparsity across data items, each record could take different time to process. We observe that operations on sparse datasets depend on the number of non-zero entries and thus if we can sample the data such that we use a *representative* sparse subset during training, we should be able to apply modeling techniques described before. However in practice, we don't see this problem as even if there is a huge skew in sparsity across rows, the skew across partitions is typically smaller.

To illustrate, we chose three of the largest sparse datasets that are part of the LibSVM repository [153, 181] and we measured the maximum number of non-zero entries present in every partition after loading the data into HDFS. We normalize these values across partitions and a CDF of partition densities is shown in Figure 3.5. We observe the the difference in sparsity between the most loaded partition and the least loaded one is less than 35% for all datasets and thus picking a random sample of partitions [168] is sufficient to model computation costs. techniques [109] which attempt to reduce skewness across partitions will also help in alleviating this problem.

3.3.3 Straggler mitigation by over-allocation

The problem of dealing with stragglers, or tasks which take much longer than other tasks is one of the main challenges in large scale data analytics [11, 58, 178]. Using cloud computing instances could further aggravate the problem due to differences in performance across instances. One technique that we use in Ernest to overcome variation among instances is to launch a small percentage of extra instances and then discard the worst performing among them before running the user's job. We use memory bandwidth and network bandwidth measurements (§3.1) to determine the slowest instances.

In our experiences with Amazon EC2 we find that even having a few extra instances can be more than sufficient in eliminating the slowest machines. To demonstrate this, we set the target cluster size as $N = 50$ `r3.2xlarge` instances and have Ernest automatically allocate a small percentage of extra nodes. We then run STREAM [122] at 30 second intervals and collect memory bandwidth measurements on all instances. Based on the memory bandwidths observed, we eliminate the slowest nodes from the cluster. Figure 3.6 shows for each allocation strategy, the CDF of the memory bandwidth obtained when picking the best N instances from all the instances allocated. We see that Ernest only needs to allocate as few as 2 (or 4$\%$) extra instances to eliminate the slowest stragglers and improve the target cluster's average memory bandwidth from 24.7 GB/s to 26 GB/s. beneficial for users with a larger budget who wish to guarantee an even higher level of cluster performance.

3.4 Ernest Discussion

In this section we look at *when* a model should be retrained and also discuss the trade-offs associated with including more fine-grained information in Ernest.

3.4.1 Model reuse

The model we build using Ernest predicts the performance for a given job for a specific dataset and a target cluster. One of the questions while using Ernest is to determine when we need to retrain the model. We consider three different circumstances here: changes in code, changes in cluster behavior and changes in data.

Code changes: If different jobs use the same dataset, the cluster and dataset remain the same, but the computation being run changes. As Ernest treats the job being run as a black-box, we will need to retrain the model for any changes to the code. This can be detected by computing hashes of the binary files.

Variation in Machine Performance: One of the concerns with using cloud computing based solutions like EC2 is that there could be performance variations over time even when a job is using the same instance types and number of instances. We investigated if this was an issue by running two machine learning jobs GLM regression and NaiveBayes repeatedly on a cluster of 64 `r3.xlarge` instances. The time taken per run of each algorithm over a 24 hour period is shown in Figure 3.7. We see that the variation over time is very small for both workloads and the standard deviation is less than 2% of the mean. Thus we believe that Ernest models should remain relevant across relatively long time periods.

Changes in datasets: As Ernest uses small samples of the data for training, the model is directly applicable as the dataset grows. When dealing with newly collected data, there are some aspects of the dataset like the number of data items per block and the number of features per data item that should remain the same for the performance properties to be similar. As some of these properties might be hard to measure, our goal is to make the model building overhead small so that Ernest can be re-run for newly collected datasets.

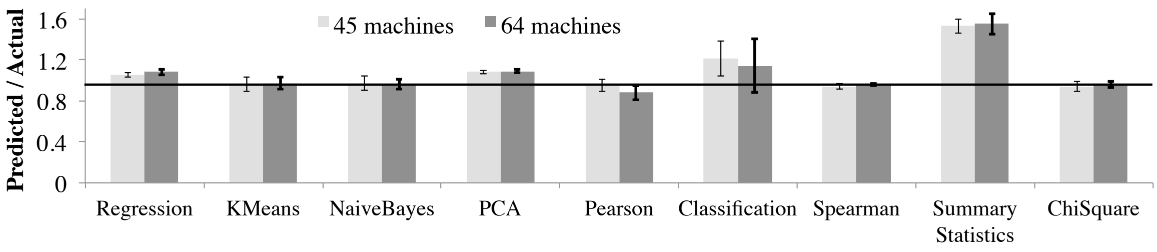
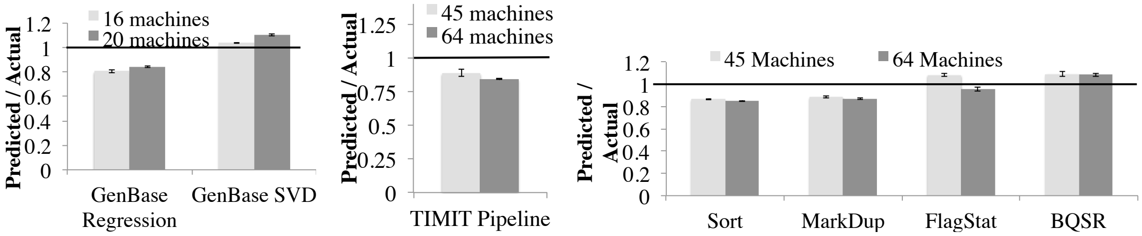


Figure 3.8: Prediction accuracy using Ernest for 9 machine learning algorithms in Spark MLlib.



(a) (b) (c)

Figure 3.9: Prediction accuracy for GenBase, TIMIT and Adam queries.

3.4.2 Using Per-Task Timings

In the model described in the previous sections, we only measure the end-to-end running time of the whole job. Existing data processing frameworks already measure fine grained metrics [79, 154] and we considered integrating task-level metrics in Ernest. One major challenge we faced here is that in the BSP model a stage only completes when its last task completes. Thus rather than predicting the average task duration, we need to estimate the maximum task duration and this requires more complex non-parametric methods like Bootstrap [62]. Further, to handle cases where the number of tasks in a stage are greater than the number of cores available, we need adapt our estimate based on the number of waves [14] of tasks. We found that there were limited gains from incorporating task-level information given the additional complexity. While we continue to study ways to incorporate new features, we found that simple features used in predicting end-to-end completion time are more robust.

3.5 Ernest Evaluation

We evaluate how well Ernest works by using two metrics: the prediction accuracy and the overhead of training for long-running machine learning jobs. In experiments where we measure accuracy, or how close a prediction is to the actual job completion time, we use the ratio of the predicted job completion time to the actual job completion time $^{\text{Predicted Time}}/_{\text{Actual Time}}$ as our metric.

The main results from our evaluation are:

- Ernest's predictive model achieves less than 20% error on most of the workloads with less than 5% overhead for long running jobs.(§3.5.2)
- Using the predictions from Ernest we can get up to $4x$ improvement in price by choosing the optimal number of instances for the speech pipeline. (§3.5.3)
- Given a training budget, experiment design improves accuracy by $30\% - 50\%$ for some workloads when compared to a cost-based approach. (§3.5.5)
- By extending the default model we are also able to accurately predict running times for sparse and randomized linear algebra operations. (§3.5.6)

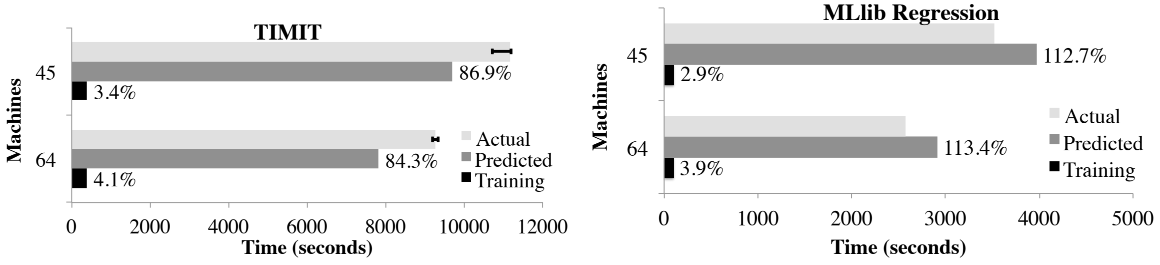
3.5.1 Workloads and Experiment Setup

We use five workloads to evaluate Ernest. Our first workload consists of 9 machine learning algorithms that are part of MLlib [124]. For algorithms designed for dense inputs, the performance characteristics are independent of the data and we use synthetically generated data with 5 million examples. We use 10K features per data point for regression, classification, clustering and 1K features for the linear algebra and statistical benchmarks.

To evaluate Ernest on sparse data, we use `splice-site` and `kdda`, two of the largest sparse classification datasets that are part of LibSVM [42]. The `splice-site` dataset contains 10M data points with around 11M features and the `kdda` dataset contains around 6.7M data points with around 20M features. To see how well Ernest performs on end-to-end pipelines, we use GenBase, ADAM and a speech recognition pipeline (§3.1). We run regression and SVD queries from GenBase on the `Large` dataset [69] (30K genes for 40K patients). For ADAM we use the high coverage NA12878 full genome from the 1000 Genomes project [1] and run four transformations: sorting, marking duplicate reads, base quality score recalibration and quality validation. The speech recognition pipeline is run on the TIMIT [90] dataset using an implementation from KeystoneML [156]. All datasets other than the one for ADAM are cached in memory before the experiments begin and we do warmup runs to trigger the JVM's just-in-time compilation. We use `r3.xlarge` machines from Amazon EC2 (each with 4 vCPUs and 30.5GB memory) unless otherwise specified. Our experiments were run with Apache Spark 1.2. Finally all our predictions were compared against at least three actual runs and the values in our graphs show the average with error bars indicating the standard deviation.

3.5.2 Accuracy and Overheads

Prediction Accuracy: We first measure the prediction accuracy of Ernest using the nine algorithms from MLlib. In this experiment we configure Ernest to use between 1 and 16 machines for training and sample between 0.1% to 10% of the dataset. We then predict the performance for cases where the algorithms use the entire dataset on 45 and 64 machines. The prediction accuracies shown in Figure 3.8 indicate that Ernest's predictions are within 12% of the actual running time for most jobs. The two exceptions where the error is higher are the `summary statistics` and



(a) TIMIT pipeline running 50 iterations

(b) MLlib Regression running 500 iterations

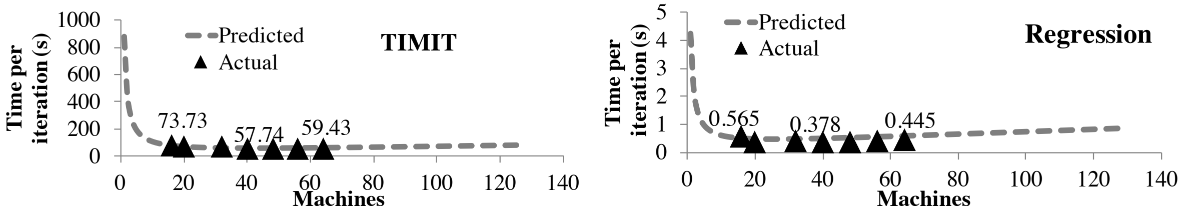
Figure 3.10: Training times vs. accuracy for TIMIT and MLlib Regression. Percentages with respect to actual running times are shown.

`glm-classification` job. In the case of `glm-classification`, we find that the training data and the actual runs have high variance (error bars in Figure 3.8 come from this) and that Ernest's prediction is within the variance of the collected data. In the case of summary statistics we have a short job where the absolute error is low: the actual running time is around 6 seconds while Ernest's prediction is around 8 seconds.

Next, we measure the prediction accuracy on GenBase and the TIMIT pipeline; the results are shown in Figure 3.9. Since the GenBase dataset is relatively small (less than 3GB in text files), we partition it into 40 splits, and restrict Ernest to use up to 6 nodes for training and predict the actual running times on 16 and 20 machines. As in the case of MLlib, we find the prediction errors to be below 20% for these workloads. Finally, the prediction accuracy for four transformations on ADAM show a similar trend and are shown in Figure 3.9(c). We note that the ADAM queries read input and write output to the *distributed filesystem* (HDFS) in these experiments and that these queries are also shuffle heavy. We find that Ernest is able to capture the I/O overheads and the reason for this is that the time to read / write a partition of data remains similar as we scale the computation.

Our goal in building Ernest is not to enforce strict SLOs but to enable low-overhead predictions that can be used to make coarse-grained decisions. We discuss how Ernest's prediction accuracy is sufficient for decisions like how many machines (§3.5.3) and what type of machines (§3.5.4) to use in the following sections.

Training Overheads: One of the main goals of Ernest is to provide performance prediction with low overhead. To measure the overhead in training we consider two long-running machine learning jobs: the TIMIT pipeline run for 50 iterations, and MLlib Regression with a mini-batch SGD solver run for 500 iterations. We configure Ernest to run 5% of the overall number of iterations during training and then linearly scale its prediction by the target number of iterations. Figures 3.10(a) and 3.10(b) show the times taken to train Ernest and the actual running times when run with 45 or 64 machines on the cluster. From the figures, we observe that for the regression problem the training time is below 4% of the actual running time and that Ernest's predictions are within 14%. For the TIMIT pipeline, the training overhead is less than 4.1% of the total running time. The



(a) (b)

Figure 3.11: Time per iteration as we vary the number of instances for the TIMIT pipeline and MLlib Regression. Time taken by actual runs are shown in the plot.

low training overhead with these applications shows that Ernest efficiently handles long-running, iterative analytics jobs.

3.5.3 Choosing optimal number of instances

When users have a fixed-time or fixed-cost budget it is often tricky to figure out how many instances should be used for a job as the communication vs. computation trade-off is hard to determine for a given workload. In this section, we use Ernest's predictions to determine the optimum number of instances. We consider two workloads from the previous section: the TIMIT pipeline and GLM regression, but here we use subsets of the full data to focus on how the job completion time varies as we increase the number of machines to 64[1]. Using the same models trained in the previous section, we predict the time taken per iteration across a wide range of number of machines (Figures 3.11(a) and 3.11(b)). We also show the actual running time to validate the predictions.

Consider a case where a user has a fixed-time budget of 1 hour (3600s) to say run 40 iterations of the TIMIT pipeline and an EC2 instance limit of 64 machines. Using Figure 3.11(a) and taking our error margin into account, Ernest is able to infer that launching 16 instances is sufficient to meet the deadline. Given that the cost of an `r3.xlarge` instance is $0.35/hour, a greedy strategy of using all the 64 machines would cost $22.4, while using the 16 machines as predicted by Ernest would only cost $5.6, a 4x difference. We also found that the 15% prediction error doesn't impact the decision as actual runs show that 15 machines is the optimum. Similarly, if the user has a budget of $15 then we can infer that using 40 machines would be faster than using 64 machines.

3.5.4 Choosing across instance types

We also apply Ernest to choose the optimal instance type for a particular workload; similar to the scenario above, we can optimize for cost given a deadline or optimize for performance given a budget. As an example of the benefits of choosing the right instance type, we re-run the TIMIT workload on three instance types (`r3.xlarge`, `r3.2xlarge` and `r3.4xlarge`) and we build

[1] We see similar scaling properties in the entire data, but we use a smaller dataset to highlight how Ernest can handle scenarios where the algorithm does not scale well.

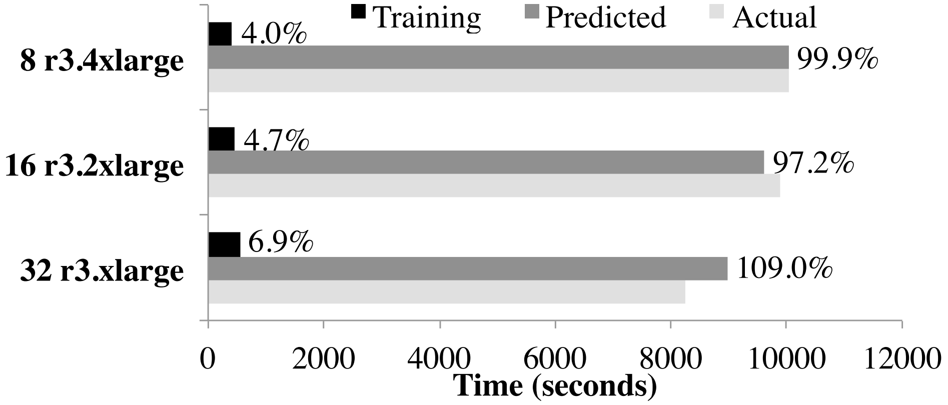
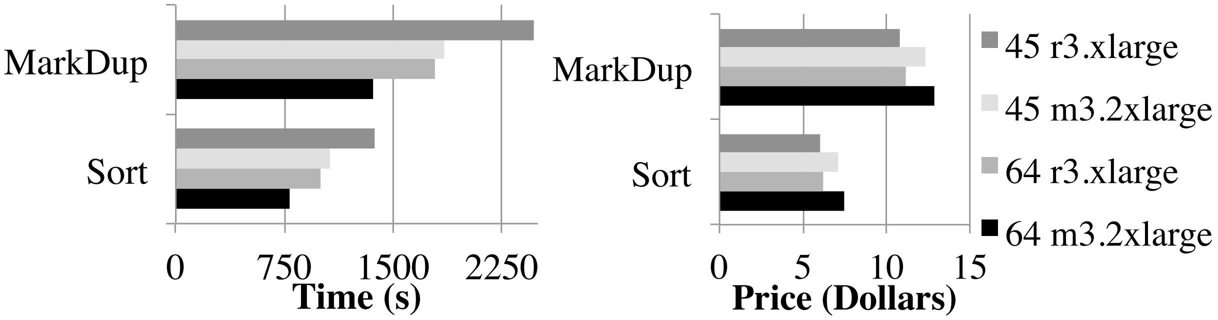


Figure 3.12: Time taken for 50 iterations of the TIMIT workload across different instance types. Percentages with respect to actual running times are shown.



(a)

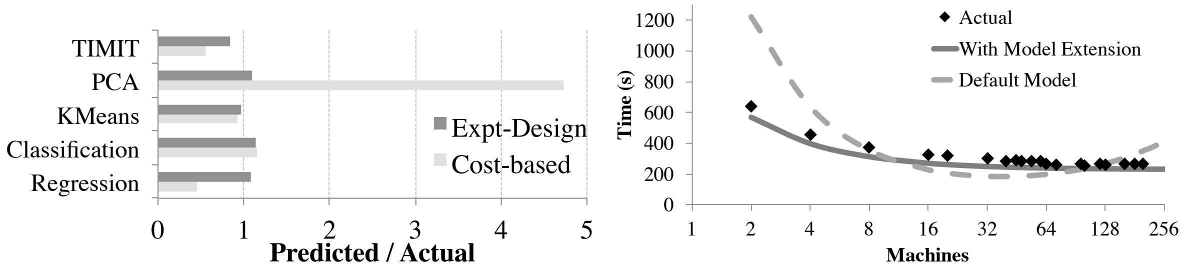
(b)

Figure 3.13: Time taken for Sort and MarkDup workloads on ADAM across different instance types.

a model for each instance type. With these three models, Ernest predicts the expected performance on *same-cost* configurations, and then picks the cheapest one. Our results (Figure 3.12) show that choosing the smaller `r3.xlarge` instances would actually be $1.2x$ faster than using the `r3.4xlarge` instances, while *incurring the same cost*. Similar to the previous section, the prediction error does not affect our decision here and Ernest's predictions choose the appropriate instance type.

We next look at how choosing the right instance type affects the performance of ADAM workloads that read and write data from disk. We compare `m3.2xlarge` instances that have two SSDs but cost $\$0.532$ per hour and `r3.xlarge` instances that have one SSD and cost $\$0.35$ an hour[2]. Results from using Ernest on 45 and 64 machines with these instance types is shown in Figure 3.13. From the we can see that using `m3.2xlarge` instances leads to better performance and that similar to the memory bandwidth analysis (§3.1.2) there are non-linear price-performance trade-offs. For example, we see that for the mark duplicates query, using 64 `m3.2xlarge` instances provides a 45% performance improvement over 45 `r3.xlarge` instances while only costing 20% more.

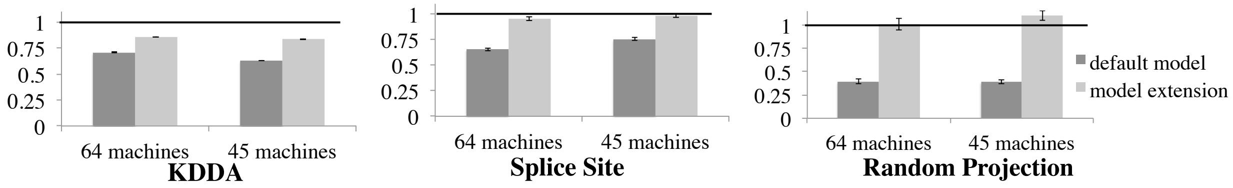
[2] Prices as of September 2015



(a) Prediction accuracy when using Ernest vs. a cost-based approach for MLlib and TIMIT workloads.

(b) Comparing KDDA models with and without extensions for different number of machines.

Figure 3.14: Ernest accuracy and model extension results.



(a)

(b)

(c)

Figure 3.15: Prediction accuracy improvements when using model extensions in Ernest. Workloads used include sparse GLM classification using KDDA, splice-site datasets and a random projection linear algebra job.

3.5.5 Experiment Design vs. Cost-based

We next evaluate the benefits of using optimal experiment design in Ernest. We compare experiment design to a greedy scheme where all the candidate training data points are sorted in increasing order of cost and we pick training points to match the cost of the points chosen in experiment design. We then train models using both configurations. A comparison of the prediction accuracy on MLlib and TIMIT workloads is shown in Figure 3.14(a).

From the figure, we note that for some workloads (e.g. KMeans) experiment design and the cost-based approach achieve similar prediction errors. However, for the Regression and TIMIT workloads, Ernest's experiment design models perform $30\% - 50\%$ better than the cost-based approach. The cost-based approach fails because when using just the cheapest training points, the training process is unable to observe how different stages of the job behave as scale and number of machines change. For example, in the case of TIMIT pipeline, the cost-based approach explores points along a weak scaling curve where *both* data size and number of machines increase, thus it is unable to model how the Solver stage scales when the amount of data is kept constant. Ernest's optimal experiment design mechanism successfully avoids this and chooses the most useful training points.

3.5.6 Model Extensions

We also measure the effectiveness of the model extensions proposed in §3.2.4 on two workloads: GLM classification run on sparse datasets (§3.3.2) and a randomized linear algebra workload that has non-linear computation time [82]. Figure 3.15 shows the prediction error for the default model and the error after the model is extended: with a $\sqrt{n}$ term for the Sparse GLM and a $\frac{nlog^2n}{mc}$ term which is the computation cost of the random projection. As we can see from the figure, using the appropriate model makes a significant difference in prediction error.

To get a better understanding of how different models can affect prediction error we use the KDDA dataset and plot the predictions from both models as we scale from 2 to 200 machines (Figure 3.14(b)). From the figure we can see that the extending the model with $\sqrt{n}$ ensures that the scaling behavior is captured accurately and that the default model can severely over-predict (at 2 machines and 200 machines) or under-predict (32 machines). Thus, while the default model in Ernest can capture a large number of workloads we can see that making simple model extensions can also help us accurately predict more complex workloads.

3.6 Ernest Conclusion

The rapid adoption of advanced analytics workloads makes it important to consider how these applications can be deployed in a cost and resource-efficient fashion. In this chapter, we studied the problem of performance prediction and show how simple models can capture computation and communication patterns. Using these models we have built Ernest, a performance prediction framework that intelligently chooses training points to provide accurate predictions with low overhead.

Chapter 4

Low-Latency Scheduling

Based on the performance model developed in Chapter 3, we see that the time taken can be divided into three main components: the time spent in parallel computation, the time spent in communication for operations like shuffle, broadcast and finally the time spent in coordination imposed by the data processing framework. We broadly classify the the computation and communication costs as being a part of the *data plane* and the coordination overheads being a part of the *control plane*.

In this chapter we consider the control plane of distributed analytics workloads and study how we can improve performance by reducing coordination overheads. These techniques are especially relevant for workloads that require low-latency execution. Examples of such workloads include iterative numerical methods (*e.g.,* stochastic gradient descent (SGD) [29] and conjugate gradient descent) and streaming data processing (*e.g.,* processing a stream of tweets [108] or updates from sensors). These workloads are characterized by stages that process a small amount of data and can execute in milliseconds. Each job is typically composed of a large number of such stages that represent iterations of the same computation.

As stages become shorter and cluster sizes increase to provide more parallelism, the overhead of centralized coordination for each stage in the BSP model becomes significant. As a result, recent work, *e.g.,* Naiad [129], Flink [144], etc., has proposed building systems with long running operators for low-latency streaming and iterative workloads. These systems typically use stateful, long-running tasks on each machine and thus eliminate the centralized coordination overheads imposed by BSP. However, this comes at a cost: since the operators are stateful, these systems must rely on complex (and expensive) techniques like distributed checkpointing [41] for fault tolerance. Furthermore, common scheduling strategies for straggler mitigation [11], data locality [14,57], etc. rely on tasks being short lived (so they can be redistributed among machines) and cannot easily be adopted for such models.

In this chapter we ask if we can meet the latency requirements of iterative and streaming applications, while still retaining the simple programming model and fault-tolerance provided by BSP. Our insight in this work is that while low latency workloads need execution granularity of few milliseconds, it is sufficient to provide scheduling decisions and fault tolerance at a coarser granularity, say every few seconds. Based on this insight, we propose Drizzle, a framework for executing

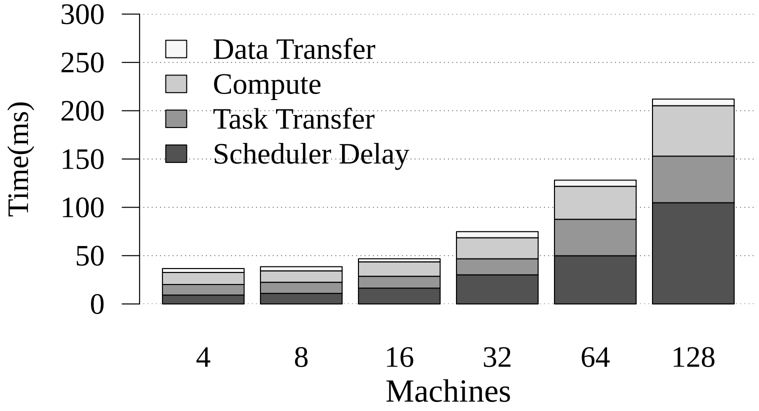


Figure 4.1: Breakdown of average time taken for task execution when running a two-stage `treeReduce` job using Spark. The time spent in scheduler delay and task transfer (which includes task serialization, deserialization, and network transfer) grows as we increase cluster size.

low latency iterative jobs that *decouples the execution and scheduling decision granularity*.

We next present the motivation for Drizzle and following that outline the design, implementation and evaluation of Drizzle on machine learning and stream processing workloads.

4.1 Case for low-latency scheduling

As an example, we consider the DAG of stages used to implement Stochastic Gradient Descent (SGD) in MLlib [124]. The job consists of a number of iterations, where each iteration first uses a map stage to compute gradients, and then uses a `treeReduce` operation to aggregate the gradients. Figure 2.4 illustrates the centralized driver's role in the execution of this job. First, the driver schedules the four tasks in the first stage of the job. To schedule a task, the driver selects a worker with available resources and sends the task to the worker. The task includes a description of the input data and a closure describing the computation the task will perform on that input data. Each of those tasks produce intermediate data to be read by tasks in the next stage, so when the tasks complete, they notify the driver and communicate metadata about the intermediate data generated. When all four tasks have completed, the driver schedules the two tasks in the next stage. The driver passes each of the two tasks information about the data generated by the previous stage, so that the tasks know where to read input data from. This process continues as the two-task stage finishes and the final one-task stage is launched and completes. At the end of an iteration the driver updates the model for the next iteration and broadcasts this value to all the workers along with the tasks of the next iteration.

Launching tasks and managing metadata are in the critical path of the job's execution, and have overheads that grow with increasing parallelism. To measure this overhead, we ran a micro-benchmark that uses a two stage `treeReduce` job modeled after one of the iterations in Figure 2.4. Each task does a very small amount of computation, to allow us to isolate the scheduler overhead. Tasks in the first stage

We use a cluster of `r3.xlarge` Amazon EC2 instances, where each machine has 4 cores, and the workload for m machines uses $4 \times m$ tasks in the first stage (i.e., the number of tasks is equal to the number of cores) and 16 tasks in the second stage. We plot the average time taken by each task for scheduling, task transfer (including serialization, deserialization, and network transfer of the task), computation[1] and data transfer. Our results in Figure 4.1 show that while the scheduling and task transfer overheads are less than 30ms at 16 machines, as we increase the cluster size the overheads grow to around 150ms (over 70% of the runtime), which is impractical for low latency workloads.

4.2 Drizzle Design

Next we detail the design of Drizzle. Drizzle builds on existing BSP-based execution model, and we show how the BSP model can be changed to dramatically reduce average scheduling and communication overheads. Reducing these overheads allows us to reduce the size of a iteration and allows us to achieve sub-second latency (of $\approx$100 ms) per iteration. In designing Drizzle, we chose to extend the BSP model since it allows us to inherit existing optimizations for high-throughput batch processing. We believe one could go in the other direction, that is start with a continuous processing system like Flink and modify it to add batching and get similar benefits.

Our high level approach to removing the overheads in the BSP-based model is to decouple the size of the iteration being processed from the interval at which coordination takes place. This decoupling will allow us to reduce the size of a iteration to achieve sub-second processing latency, while ensuring that coordination, which helps the system adapt to failures and cluster membership changes, takes place every few seconds. We focus on the two sources of coordination that exists in BSP systems. First we look at the centralized coordination that exists between iterations and how we can remove this with group scheduling. Following that, we discuss how we can remove the barrier within a iteration using pre-scheduling.

4.2.1 Group Scheduling

BSP frameworks like Spark, FlumeJava [39] or Scope [37] use centralized schedulers that implement many complex scheduling techniques; these include: algorithms to account for locality [183], straggler mitigation [11], fair sharing [93] etc. Scheduling a single stage in these systems proceeds as follows: first, a centralized scheduler computes which worker each task should be assigned to, taking in the account locality and other constraints. Following this tasks are serialized and sent to workers using an RPC. The complexity of the scheduling algorithms used coupled with computational and network limitations of the single machine running the scheduler imposes a fundamental limit on how fast tasks can be scheduled.

[1] The computation time for each task increases with the number of machines even though the first stage computation remains the same. This is because tasks in second stage have to process more data as we increase the number of machines.

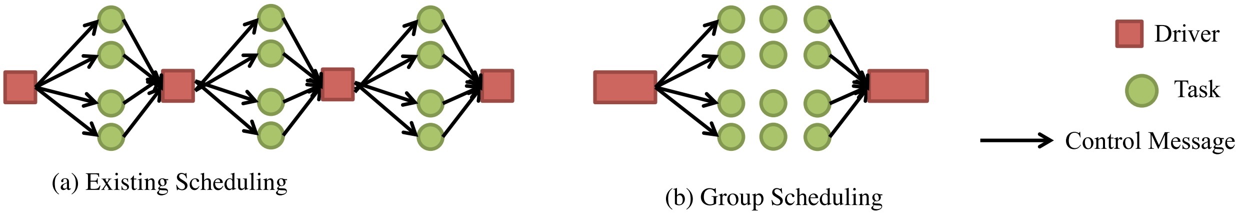


Figure 4.2: Group scheduling amortizes the scheduling overheads across multiple iterations of a streaming job.

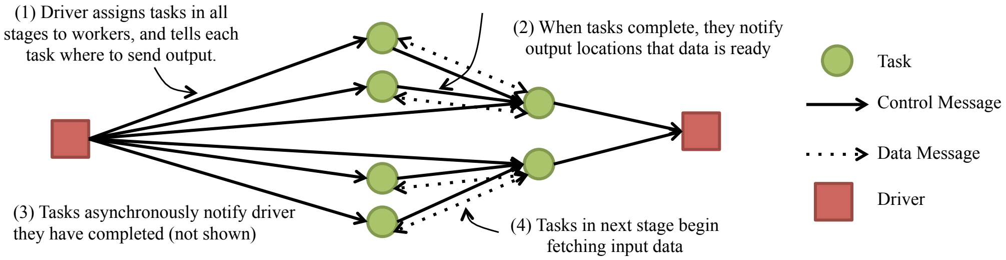

(1) Driver assigns tasks in all stages to workers, and tells each task where to send output.
(2) When tasks complete, they notify output locations that data is ready
(3) Tasks asynchronously notify driver they have completed (not shown)
(4) Tasks in next stage begin fetching input data
Task
Control Message
Data Message
Driver

Figure 4.3: Using pre-scheduling, execution of a iteration that has two stages: the first with 4 tasks; the next with 2 tasks. The driver launches all stages at the beginning (with information about where output data should be sent to) so that executors can exchange data without contacting the driver.

The limitation a single centralized scheduler has been identified before in efforts like Sparrow [136] and Apollo [30]. However, these systems restrict themselves to cases where scheduled tasks are independent of each other, e.g., Sparrow forwards each scheduling request to one of many distributed schedulers which do not coordinate among themselves and hence cannot account for dependencies between requests. In Drizzle, we focus on iterative jobs where there are dependencies between batches and thus coordination is required when scheduling parts of a single job. To alleviate centralized scheduling overheads, we study the execution DAG of iterative jobs.

We observe that in iterative jobs, the computation DAG used to process iterations is largely static, and changes infrequently. Based on this observation, we propose reusing *scheduling decisions* across iterations. Reusing scheduling decisions means that we can schedule tasks for *multiple iterations* (or a group) at once (Figure 4.2) and thus amortize the centralized scheduling overheads. To see how this can help, consider a job used to compute moving averages. Assuming the data sources remain the same, the locality preferences computed for every iteration will be same. If the cluster configuration also remains static, the same worker to task mapping will be computed for every iteration. Thus we can run scheduling algorithms once and reuse its decisions. Similarly, we can reduce network overhead of RPCs by combining tasks from multiple iterations into a single message.

When using group scheduling, one needs to be careful in choosing how many iterations are scheduled at a time. We discuss how the group size affects the performance and adaptivity properties in §4.2.3 and present techniques for automatically choosing this size in §4.2.4.

4.2.2 Pre-Scheduling Shuffles

While the previous section described how we can eliminate the barrier or coordination between iterations, as described in Section 3.1 (Figure 2.4), existing BSP systems also impose a barrier within a iteration to coordinate data transfer for shuffle operations. We next discuss how we can eliminate these barriers as well and thus eliminate all coordination within a group.

In a shuffle operation we have a set of upstream tasks (or map tasks) that produce output and a set of downstream tasks (or reduce tasks) that receive the outputs and run the reduction function. In existing BSP systems like Spark or Hadoop, upstream tasks typically write their output to local disk and notify the centralized driver of the allocation of output records to the different reducers. The driver then applies task placement techniques [49] to minimize network overheads and creates downstream tasks that pull data from the upstream tasks. Thus in this case the metadata is communicated through the centralized driver and then following a barrier, the data transfer happens using a pull based mechanism.

To remove this barrier, we *pre-schedule downstream tasks* before the upstream tasks (Figure 4.3) in Drizzle. We perform scheduling so downstream tasks are launched first; this way upstream tasks are scheduled with metadata that tells them which machines running the downstream tasks need to be notified on completion. Thus, data is directly transferred between workers without any centralized coordination. This approach has two benefits. First, it scales better with the number of workers as it avoids centralized metadata management. Second, it removes the barrier, where succeeding stages are launched only when all the tasks in the preceding stage complete.

We implement pre-scheduling by adding a local scheduler on each worker machine that manages pre-scheduled tasks. When pre-scheduled tasks are first launched, these tasks are marked as inactive and do not use any resources. The local scheduler on the worker machine tracks the data dependencies that need to be satisfied. When an upstream task finishes, it materializes the output on local disk, notifies the corresponding downstream workers and asynchronously notifies the centralized scheduler. The local scheduler at the downstream task then updates the list of outstanding dependencies. When all the data dependencies for an inactive task have been met, the local scheduler makes the task active and runs it. When the task is run, it fetches the files materialized by the upstream tasks and continues processing. Thus we implement a push-metadata, pull-based data approach that minimizes the time to trigger tasks while allowing the downstream tasks to control when the data is transferred.

4.2.3 Adaptivity in Drizzle

Group scheduling and shuffle pre-scheduling eliminate barriers both within and across iterations and ensure that barriers occur only once every group. However in doing so, we incur overheads when adapting to changes and we discuss how this affects fault tolerance, elasticity and workload changes below. This overhead largely does not affect record processing latency, which continues to happen within a group.

Fault tolerance. Similar to existing BSP systems we create synchronous checkpoints at regular intervals in Drizzle. The checkpoints can be taken at the end of any iteration and the end of a group of iterations presents one natural boundary. We use heartbeats from the workers to the centralized scheduler to detect machine failures. Upon noticing a failure, the scheduler resubmits tasks that were running on the failed machines. By default these recovery tasks begin execution from the latest checkpoint available. As the computation for each iteration is deterministic we further speed up the recovery process with two techniques. First, recovery tasks are executed in parallel [187] across many machines. Second, we also reuse any intermediate data that was created by map stages run in earlier iterations. This is implemented with lineage tracking, a feature that is already present in existing BSP systems.

Using pre-scheduling means that there are some additional cases we need to handle during fault recovery in Drizzle. For reduce tasks that are run on a new machine, the centralized scheduler pre-populates the list of data dependencies that have been completed before. This list is maintained based on the asynchronous updates from upstream tasks. Similarly the scheduler also updates the active upstream (map) tasks to send outputs for succeeding iterations to the new machines. In both cases, if the tasks encounter a failure in either sending or fetching outputs they forward the failure to the centralized scheduler. Thus we find having a centralized scheduler simplifies design and helps ensure that there is a single source that workers can rely on to make progress.

Elasticity. In addition to handling nodes being removed, we can also handle nodes being added to improve performance. To do this we integrate with existing cluster managers such as YARN [17] and Mesos [88] and the application layer can choose policies [54] on when to request or relinquish resources. At the end of a group boundary, Drizzle updates the list of available resources and adjusts the tasks to be scheduled for the next group. Thus in this case, using a larger group size could lead to larger delays in responding to cluster changes.

4.2.4 Automatically selecting group size

Intuitively, the group size controls the performance to co-ordination trade-off in Drizzle. The total runtime of the job can be split between time spent in coordination and time spent in data-processing. In the absence of failures, the job's running time is minimized by avoiding coordination, while more frequent coordination enables better adaptability. These two objectives are thus at odds with each other. In Drizzle, we explore the trade-off between these objectives by bounding coordination overheads while maximizing adaptability. Thus, we aim to choose a group size that is the smallest possible while having a fixed bound on coordination overheads.

We implement an adaptive group-size tuning algorithm that is inspired by TCP congestion control [31]. During the execution of a group we use counters to track the amount of time spent in various parts of the system. Using these counters we are then able to determine what fraction of the end-to-end execution time was spent in scheduling and other coordination vs. time spent on the workers executing tasks. The ratio of time spent in scheduling to the overall execution gives us the scheduling overhead and we aim to maintain this overhead within user specified lower and upper bounds.

When the overhead goes above the upper bound, we multiplicatively increase the group size to ensure that the overhead decreases rapidly. Once the overhead goes below the lower bound, we additively decrease the group size to improve adaptivity. This is analogous to applying AIMD policy to determine the coordination frequency for a job. AIMD is widely used in TCP, and has been shown to provably converge in general [94]. We use exponentially averaged scheduling overhead measurements when tuning group size. This ensures that we are stable despite transient latency spikes from garbage collection and other similar events.

The adaptive group-size tuning algorithms presents a simple approach that handles the most common scenarios we find in large scale deployments. However the scheme requires users to provide upper and lower bounds for the coordination overhead and these bounds could be hard to determine in a new execution environment. In the future we plan to study techniques that can measure various aspects of the environment to automatically determine the scheduling efficiency.

4.2.5 Conflict-Free Shared Variables

An additional challenge for machine learning workloads is how to track and disseminate updates to a shared model with minimal coordination. We next describe conflict-free shared variables, an extension that enables light-weight data sharing in Drizzle.

Most machine learning algorithms perform commutative updates to model variables [113], and sharing model updates across workers is equivalent to implementing a `AllReduce` [189]. To enable commutative updates *within* a group of iterations we introduce conflict-free shared variables.

Conflict-free shared variables are similar in spirit to CRDTs [148] and provide an API to access and commutatively update shared data. We develop an API that can be used with various underlying implementations. Our API consists of two main functions: (1) A `get` function that optionally blocks to synchronize updates. This is typically called at the beginning of a task. In synchronous mode, it waits for all updates from the previous iterations. (2) A `commit` function that specifies that a task has finished all the updates to the shared variable. Additionally we provide callbacks to the shared variable implementation when a group begins and ends. This is to enable each implementation to checkpoint their state at group boundaries and thus conflict-free shared variables are a part of the unified fault tolerance strategy in Drizzle.

In our current implementation we focus on models that can fit on a single machine (these could still be many millions of parameters given a standard server has $\approx$ 200GB memory) and build support for replicated shared variables with synchronous updates. We implement a merge strategy that aggregates all the updates on a machine before exchanging updates with other replicas. While other parameter servers [56, 114] implement more powerful consistency semantics, our main focus here is to study how the control plane overheads impact performance. We plan to integrate Drizzle with existing state-of-the-art parameter servers [113, 176] in the future.

4.2.6 Data-plane Optimizations for SQL

The previous sections describe the design of the control-plane used in Drizzle, next we describe some of data plane optimizations enabled by Drizzle. We specifically focus on stream processing workloads implemented in systems like Spark streaming and study the importance of batching for the data plane. We start by analyzing the query workload for a popular cloud hosted data analytics

Aggregate	Percentage of Queries
Count	60.55
First/Last	25.9
Sum/Min/Max	8.640
User Defined Function	0.002
Other	4.908

Table 4.1: Breakdown of aggregations used in a workload containing over 900,000 SQL and streaming queries.

provider. We use this analysis to motivate the need for efficient support for aggregations and describe how batching can provide better throughput and latency for such workloads.

Workload analysis. We analyze over 900,000 SQL queries and streaming queries executed on a popular cloud-based data analytics platform. These queries were executed by different users on a variety of data sets. We parsed these queries to determine the set of frequently used operators: we found that around 25% of queries used one or more aggregation functions. While our dataset did not explicitly distinguish between streaming and ad-hoc SQL queries, we believe that streaming queries which update dashboards naturally require aggregations across time. This platform supports two kinds of aggregation functions: aggregations that can use partial merge operations (e.g., `sum`) where the merge operation can be distributed across workers and complete aggregations (e.g., `median`) which require data to be collected and processed on a single machine. We found that over 95% of aggregation queries only made use of aggregates supporting partial merges. A complete breakdown is shown in Table 4.1. In summary, our analysis shows that supporting efficient computation of partial aggregates like `count`, `sum` is important for achieving good performance.

Optimization within a batch. In order to support aggregates efficiently, batching the computation of aggregates can provide significant performance benefits. These benefits come from two sources: using vectorized operations on CPUs [27] and by minimizing network traffic from partial merge operations [20]. For example to compute a count of how many ad-clicks were generated for each publisher, we can aggregate the counts per publisher on each machine during a `map` and combine them during the `reduce` phase. We incorporate optimizations within a batch in Drizzle and measure the benefits from this in Section §4.4.5.

Optimization across batches and queries. Using Drizzle's architecture also enables optimizations across iterations. This is typically useful in case the query plan needs to be changed due to changes in the data distribution. To enable such optimizations, during every iteration, a number of metrics about the execution are collected. These metrics are aggregated at the end of a group and passed on to a query optimizer [20, 128] to determine if an alternate query plan would perform better. Finally, a BSP based architecture also enables reuse of intermediate results *across streaming queries*. This could be useful in scenarios where a number of streaming queries are run on the same dataset and we plan to investigate the benefits from this in the future.

4.2.7 Drizzle Discussion

We discuss other design approaches to group scheduling and extensions to pre-scheduling that can further improve performance.

Other design approaches. An alternative design approach we considered was to treat the existing scheduler in BSP systems as a black-box and pipeline scheduling of one iteration with task execution of the previous iteration. That is, while the first iteration executes, the centralized driver schedules one or more of the succeeding iterations. With pipelined scheduling, if the execution time for a iteration is t_{exec} and scheduling overhead is t_{sched}, then the overall running time for b iterations is $b \times max(t_{exec}, t_{sched})$. The baseline would take $b \times (t_{exec} + t_{sched})$. We found that this approach is insufficient for larger cluster sizes, where the value of t_{sched} can be greater than t_{exec}.

Another design approach we considered was to model task scheduling as leases [76] that could revoked if the centralized scheduler wished to make any changes to the execution plan. By adjusting the lease duration we can similarly control the coordination overheads. However using this approach would require reimplementing the execution model used by BSP-style systems and we found that group scheduling offered similar behavior while providing easier integration with existing systems.

Improving Pre-Scheduling. While using pre-scheduling in Drizzle, the reduce tasks usually need to wait for a notification from all upstream map tasks before starting execution. We can reduce the number of inputs a task waits for if we have sufficient semantic information to determine the communication structure for a stage. For example, if we are aggregating information using binary tree reduction, each reduce task only requires the output from two map tasks run in the previous stage. In general inferring the communication structure of the DAG that is going to be generated is a hard problem because of user-defined map and hash partitioning functions. For some high-level operators like `treeReduce` or `broadcast` the DAG structure is predictable. We have implemented support for using the DAG structure for `treeReduce` in Drizzle and plan to investigate other operations in the future.

Quality of Scheduling. Using group and pre-scheduling requires some minor modifications to the scheduling algorithm used by the underlying systems. The main effect this introduces for scheduling quality is that the scheduler's decision algorithm is only executed once for each group of tasks. This coarser scheduling granularity can impact algorithms like fair sharing, but this impact is bounded by the size of a group, which in our experience is limited to a few seconds at most. Further, while using pre-scheduling the scheduler is unaware of the size of the data produced from the upstream tasks and thus techniques to optimize data transfer [49] cannot be applied. Similarly database-style optimizations that perform dynamic rebalancing [103] of tasks usually depend on data statistics and cannot be used within a group. However for streaming and machine learning applications we see that the data characteristics change over the course of seconds to minutes and thus we can still apply these techniques using previously collected data.

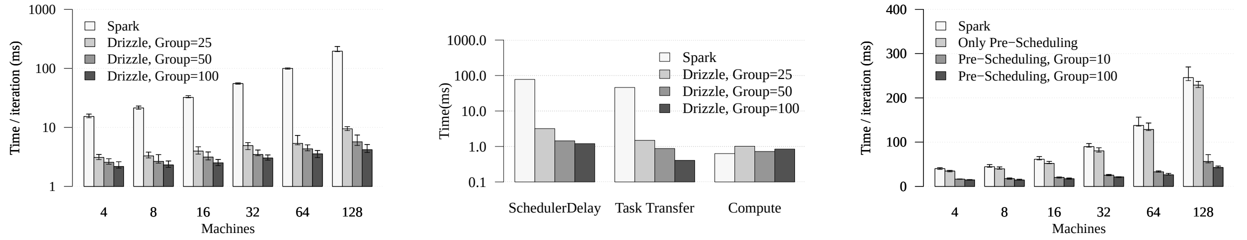
4.3 Drizzle Implementation

We have implemented Drizzle by extending Apache Spark 2.0.0. Our implementation required around 4000 lines of Scala code changes to the Spark scheduler and execution engine. We next describe some of the additional performance improvements we made in our implementation and also describe how we integrated Drizzle with Spark Streaming and MLlib.

Spark Improvements. The existing Spark scheduler implementation uses two threads: one to compute the stage dependencies and locality preferences, and the other to manage task queuing, serializing, and launching tasks on executors. We observed that when several stages pre-scheduled together, task serialization and launch is often a bottleneck. In our implementation we separated serializing and launching tasks to a dedicated thread and optionally use multiple threads if there are many stages that can be scheduled in parallel. We also optimized locality lookup for pre-scheduled tasks and these optimizations help reduce the overheads when scheduling many stages in advance. However there are other sources of performance improvements we have not yet implemented in Drizzle. For example, while iterative jobs often share the same closure across iterations we do not amortize the closure serialization across iterations. This requires analysis of the Scala byte code that is part of the closure and we plan to explore this in the future.

MLlib. To integrate machine learning algorithms with Drizzle we introduce a new iteration primitive (similar to `iterate` in Flink), which allows developers to specify the code that needs to be run on every iteration. We change the model variables to be stored using conflict-free shared variables (instead of broadcast variables) and similarly change the gradient computation functions. We use sparse and dense vector implementations of conflict-free shared variables that are a part of Drizzle to implement gradient-descent based optimization algorithms. We plan to support other algorithms like dual coordinate ascent [147] in the future. Higher level algorithms like Logistic Regression, SVM require no change as they can directly use the modified optimization algorithms.

Spark Streaming. The Spark Streaming architecture consists of a `JobGenerator` that creates a Spark RDD and a closure that operates on the RDD when processing a micro-batch. Every micro-batch in Spark Streaming is associated with an execution timestamp and therefore each generated RDD has an associated timestamp. In Drizzle, we extend the `JobGenerator` to submit a number of RDDs at the same time, where each generated RDD has its appropriate timestamp. For example, when Drizzle is configured to use group size of 3, and the starting timestamp is t, we would generate RDDs with timestamps t, $t+1$ and $t+2$. One of the challenges in this case lies in how to handle input sources like Kafka [106], HDFS etc. In the existing Spark architecture, the metadata management of which keys or files to process in a micro-batch is done by the centralized driver. To handle this without centralized coordination, we modified the input sources to compute the metadata on the workers as a part of the tasks that read input. Finally, we note these changes are restricted to the Spark Streaming implementation and user applications do not need modification to work with Drizzle.



(a) Time taken for a single stage job with 100 iterations while varying the group size. With group size of 100, Drizzle takes less than 5ms per iteration.

(b) Breakdown of time taken by a task in the single stage micro-benchmark when using 128 machines. We see that Drizzle can lower the scheduling overheads.

(c) Time taken per iteration for a streaming job with shuffles. We break down the gains between pre-scheduling and group scheduling.

Figure 4.4: Micro-benchmarks for performance improvements from group scheduling and pre-scheduling.

4.4 Drizzle Evaluation

We next evaluate the performance of Drizzle. First, we use a series of microbenchmarks to measure the scheduling performance of Drizzle and breakdown the time taken at each step in the scheduler. Next we measure the impact of using Drizzle with two real world applications: a logistic regression task on a standard machine learning benchmark and an industrial streaming benchmark [179]. Finally, we evaluate the adaptability of Drizzle using a number of different scenarios including fault tolerance, elasticity and evaluate our auto-tuning algorithm. We compare Drizzle to Apache Spark, a BSP style-system and Apache Flink, a record-at-time stream processing system.

4.4.1 Setup

We ran our experiments on 128 `r3.xlarge` instances in Amazon EC2. Each machine has 4 cores, 30.5 GB of memory and 80 GB of SSD storage. We configured Drizzle to use 4 slots for executing tasks on each machine. For all our experiments we warm up the JVM before taking measurements. We use Apache Spark v2.0.0 and Apache Flink v1.1.1 as baselines for our experiments. All the three systems we compare are implemented on the JVM and we use the same JVM heap size and garbage collection flags for all of them.

4.4.2 Micro benchmarks

In this section we present micro-benchmarks that evaluate the benefits of group scheduling and pre-scheduling. We run ten trials for each of our micro-benchmarks and report the median, 5th and 95th percentiles.

Group Scheduling

We first evaluate the benefits of group scheduling in Drizzle, and its impact in reducing scheduling overheads with growing cluster size. We use a weak scaling experiment where the amount of computation per task is fixed (we study strong scaling in §4.4.3) but the size of the cluster (and hence number of tasks) grow. For this experiment, we set the number of tasks to be equal to the number of cores in the cluster. In an ideal system the computation time should remain constant. We use a simple workload where each task computes the sum of random numbers and the computation time for each iteration is less than 1ms. Note that there are no shuffle operations in this benchmark. We measure the average time taken per iteration while running 100 iterations and we scale the cluster size from 4–128 machines.

As discussed in §3.1, we see that (Figure 4.4(a)) task scheduling overheads grow for Spark as the cluster size increases and is around 200ms when using 128 machines. Drizzle is able to amortize these overheads leading to a $7 - 46\times$ speedup across cluster sizes. With a group size of 100 and 128 machines, Drizzle has scheduler overhead of less than 5ms compared to around 195ms for Spark.

We breakdown where the benefits come from in Figure 4.4(b). To do this we plot the average time taken by each task for scheduling, task transfer (including serialization, deserialization, and network transfer of the task) and computation. We can see that scheduling and task transfer dominate the computation time for this benchmark with Spark and that Drizzle is able to amortize both of these using group scheduling.

Pre-Scheduling Shuffles

To measure benefits from pre-scheduling we use the same setup as in the previous subsection but add a shuffle stage to every iteration with 16 reduce tasks. We compare the time taken per iteration while running 100 iterations in Figure 4.4(c). Drizzle achieves between $2.7x$ to $5.5x$ speedup over Spark as we vary cluster size.

Figure 4.4(c) also shows the benefits of just using pre-scheduling (i.e., group size $= 1$). In this case, we still have barriers between the iterations and only eliminate barriers within a single iteration. We see that the benefits from just pre-scheduling are limited to only 20ms when using 128 machines. However for group scheduling to be effective we need to pre-schedule all of the tasks in the DAG and thus pre-scheduling is essential.

Finally, we see that the time per iteration of the two-stage job (45ms for 128 machines) is significantly higher than the time per iteration of the one-stage job in the previous section (5 ms). While part of this overhead is from messages used to trigger the reduce tasks, we also observe that the time to fetch and process the shuffle data in the reduce task grows as the number of map tasks increase. To reduce this overhead, we plan to investigate techniques that reduce connection initialization time and improve data serialization/deserialization [123].

Conflict-free Shared Variables

To evaluate the performance of conflict-free shared variables we run an iterative workload where we perform an `AllReduce` operation in each iteration. The aggregate value is used to start the next iteration. For our baseline, we implement the same operation in Spark using a reduction to the driver followed by a broadcast to all the workers. We measure the performance trends as we vary the (a) size of the shared variable (b) number of machines used and (c) the group size.

For a smaller vector size of 32k (Figure 4.5(a)) we see that using conflict-free shared variables alone (i.e.,group size = 1) does not yield much performance improvements. However in this case where the data plane operations are very cheap we find that using group scheduling can improve performance by $5x$ across various cluster sizes.

For larger vector size of 1MB we see that the data plane becomes the bottleneck especially for larger clusters. In this case conflict-free shared variables give up to $4x$ improvement and most of this improvement comes from eagerly aggregating updates on a single machine before sending them to other machines. Finally, we also note that group scheduling becomes less important when the data plane overheads are high and we also study this in Section 4.4.3 with a real-world machine learning benchmark.

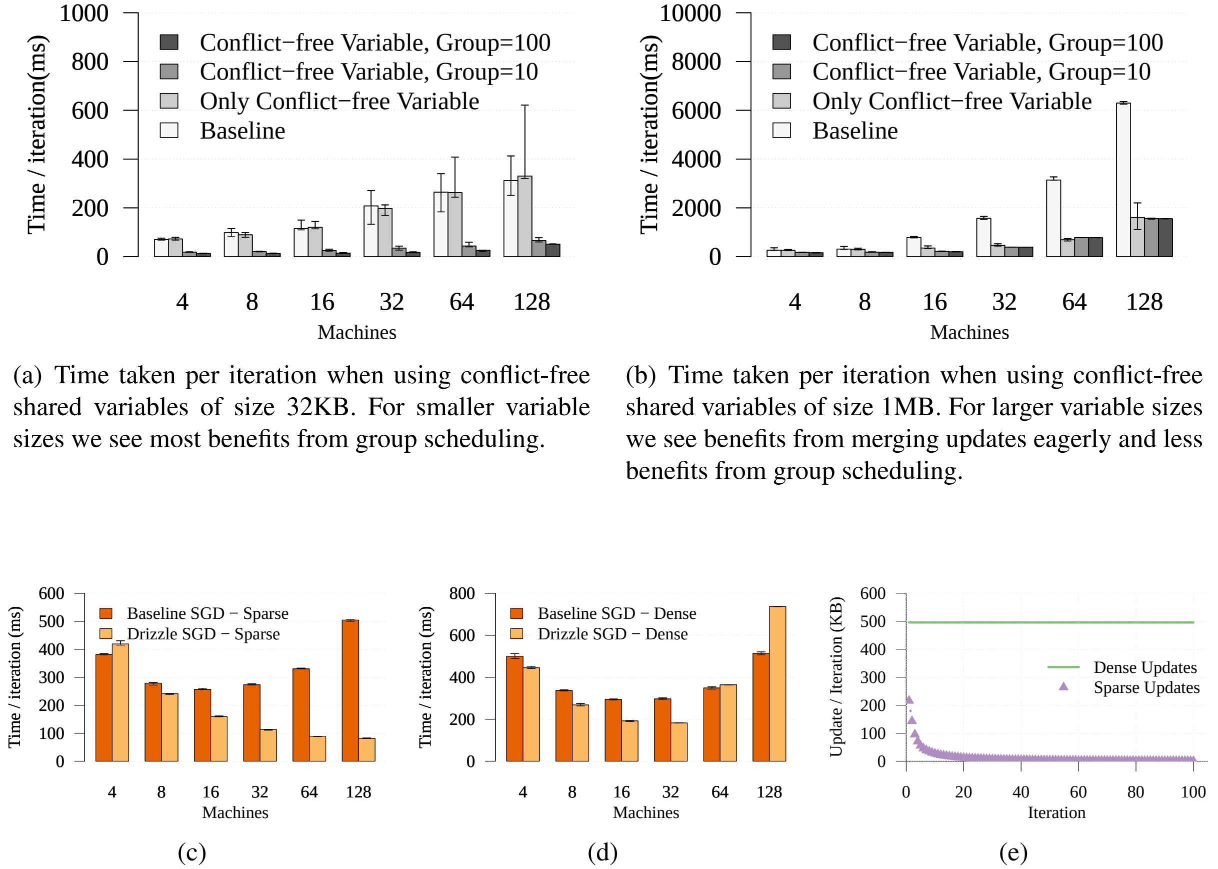
Varying vector size. Varying the size of the dense vector changes the amount of data that is transferred during the `AllReduce` operation. For small updates of 32 bytes or 1KB we see that Drizzle is around $6x$ faster than the baseline. This comes from both avoiding scheduling overheads for each iteration and by efficient aggregation of commutative updates. As we increase the vector size we see that the scheduling overheads become smaller and the network bandwidth of the centralized aggregator in Drizzle becomes the bottleneck. We still see around $4x$ win over the baseline as Drizzle aggregates updates from all the tasks on the machine (our machines have 4 cores) while Spark sends each task result separately.

Varying batch size. Finally we fix the cluster size (128 machines), vector size (1KB) and measure the importance of the batch size. Similar to §4.4.2 we see that using a batch size of 25 or greater is sufficient to amortize the scheduling delay in this case.

4.4.3 Machine Learning workloads

Next, we look at Drizzle's benefits for machine learning workloads. We evaluate this with logistic regression implemented using stochastic gradient descent (SGD). We use the Reuters Corpus Volume 1 (RCV1) [112] dataset, which contains cosine-normalized TF-IDF vectors from text documents. We use SGD to train two models from this data: the first is stored as a dense vector, while for the second we use L1-regularization [161] with thresholding [114] to get a sparse model. We show the time taken per iteration for Drizzle when compared to Spark in Figure 4.5.

From the figure we see that while using dense updates, both Drizzle and Spark hit a scaling limit at around 32 machines. This is because beyond 32 machines the time spent in communicating model updates dominates the time spent computing gradients. However using sparse updates we find that Drizzle can scale better and the time per-iteration drops to around 80ms at 128 machines (compared to 500ms with Spark). To understand how sparse updates help, we plot the size of data



(a) Time taken per iteration when using conflict-free shared variables of size 32KB. For smaller variable sizes we see most benefits from group scheduling.

(b) Time taken per iteration when using conflict-free shared variables of size 1MB. For larger variable sizes we see benefits from merging updates eagerly and less benefits from group scheduling.

(c)

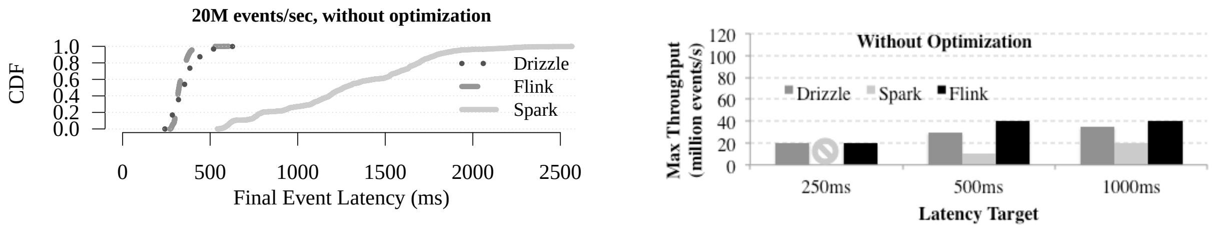
(d)

(e)

Figure 4.5: Time taken per iteration of Stochastic Gradient Descent (SGD) run on the RCV1 dataset. We see that using sparse updates, Drizzle can scale better as the cluster size increases.

transferred per-iteration in Figure 4.5(e). In addition to transmitting less data, we find that as the model gets closer to convergence fewer features get updated, further reducing the communication overhead. While using sparse updates with Spark, though the model updates are small, the task scheduling overheads dominate the time taken per iteration and thus using more than 32 machines makes the job run slower.

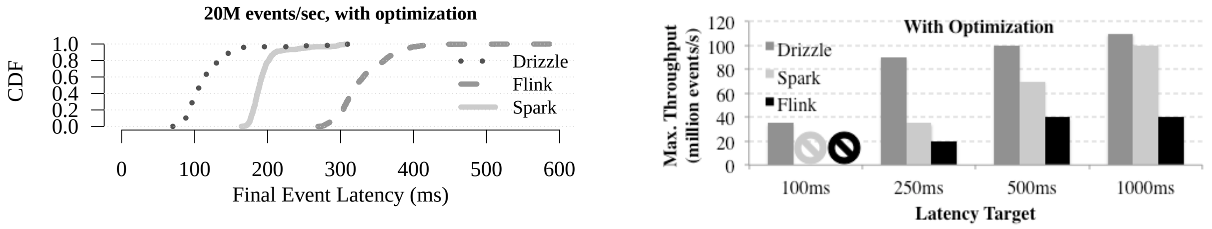
Finally, given the computation and communication trade-offs above, we note that using a large cluster is not efficient for this dataset [123]. In fact, RCV1 can be solved on a single machine using tools like LibLinear [64]. However, using a smaller dataset stresses the scheduling layers of Drizzle and highlights where the scaling bottlenecks are. Further, for problems [113] that do not fit on a single machine, algorithms typically sample the data in each iteration and our example is similar to using a sample size of 677k for such problems.



(a) CDF of event processing latencies when using `groupBy` operations in the micro-batch model. Drizzle matches the latencies of Flink and is around 3.6x faster than Spark.

(b) Maximum throughput achievable at a given latency target by Drizzle, Spark and Flink. Spark is unable to sustain latency of 250ms while Drizzle and Flink achieve similar throughput.

Figure 4.6: Latency and throughput comparison of Drizzle with Spark and Flink on the Yahoo Streaming benchmark.



(a) **CDF of event processing latencies when using micro-batch optimizations with the Yahoo Streaming Benchmark. Drizzle is 2x faster than Spark and 3x faster than Flink.**

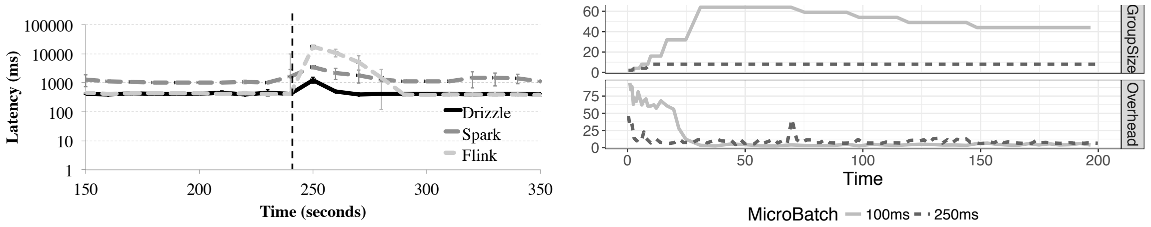
(b) Maximum throughput achievable at a given latency target by Drizzle, Spark and Flink when using micro-batch optimizations. Spark and Flink fail to meet the 100ms latency target and Drizzle's throughput increases by 2–3x by optimizing execution within a micro-batch.

Figure 4.7: Effect of micro-batch optimization in Drizzle in terms of latency and throughput.

4.4.4 Streaming workloads

Next, we demonstrate Drizzle's benefits for streaming applications. Each *micro-batch* in the streaming application is represented as a single iteration of computation to Drizzle. We use the Yahoo! streaming benchmark [179] which mimics running analytics on a stream of ad-impressions. A producer inserts JSON records into a stream. The benchmark then groups events into 10-second windows per ad-campaign and measures how long it takes for all events in the window to be processed after the window has ended. For example if a window ended at time a and the *last event* from the window was processed at time b, the processing latency for this window is said to be $b-a$.

When implemented using the micro-batch model in Spark and Drizzle, this workload consists of two stages per micro-batch: a map-stage that reads the input JSONs and buckets events into a window and a reduce stage that aggregates events in the same window. For the Flink implementation we use the optimized version [55] which creates windows in Flink using the event timestamp that is present in the JSON. Similar to the micro-batch model we have two operators here, a map



(a) Event Latency for the Yahoo Streaming Benchmark when handling failures. At 240 seconds, we kill one machine in the cluster. Drizzle has lower latency during recovery and recovers faster.

(b) Behavior of group size auto tuning with the Yahoo Streaming benchmark when using two different micro-batch sizes. The optimal group size is lower for the larger micro-batch.

Figure 4.8: Behavior of Drizzle across streaming benchmarks and how the group size auto-tuning behaves for the Yahoo streaming benchmark.

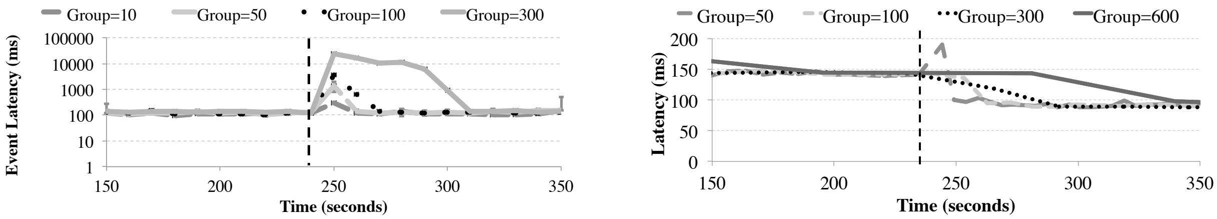
operator that parses events and a window operator that collects events from the same window and triggers an update every 10 seconds.

For our evaluation, we created an input stream that inserts JSON events and measure the event processing latency. We use the first five minutes to warm up the system and report results from the next five minutes of execution. We tuned each system to minimize latency while meeting throughput requirements. In Spark this required tuning the micro-batch size, while in Flink we tuned the buffer flush duration.

Latency. The CDF of processing latencies for 20M events/second on 128 machines is shown in Figure 4.6(a). We see Drizzle is able to achieve a median latency of around 350ms and matches the latency achieved by Flink, a continuous operator streaming system. We also verified that the latency we get from Flink match previously reported numbers [55, 179] on the same benchmark. We also see that by reducing scheduling overheads, Drizzle gets around 3.6x lower median latency than Spark.

Throughput. We next compare the maximum throughput that can be achieved given a latency target. We use the Yahoo Streaming benchmark for this and for Spark and Drizzle we set the latency target by adjusting the micro-batch size and measure the maximum throughput that can be sustained in a stable fashion. For continuous operator systems like Flink there is no direct way to configure a latency target. Thus, we measure the latency achieved as we increase the throughput and report the maximum throughput achieved within our latency bound. The results from this experiment are shown in Figure 4.6(b). From the figure we can see that while Spark crashes at very low latency target of 250ms, Drizzle and Flink both get around 20M events/s. At higher latency targets we find that Drizzle gets 1.5-3x more throughput than Spark and that this number reduces as the latency target grows. This is because the overheads from scheduling become less important at higher latency targets and thus the benefits from Drizzle become less relevant.

Fault Tolerance. We use the same Yahoo streaming benchmark as above and benchmark the fault tolerance properties of all three systems. In this experiment we kill one machine in the cluster after 240 seconds and measure the event processing latency as before. Figure 4.8(a) shows the



(a) Event Latency for the Yahoo Streaming Benchmark when handling failures. At 240 seconds, we kill one machine in the cluster and we see that having a smaller group allows us to react faster to this.

(b) Average time taken per micro-batch as we change the number of machines used. At 240 seconds, we increase the number of machines from 64 to 128. Having a smaller group allows us to react faster to this.

Figure 4.9: Effect of varying group size in Drizzle.

latency measured for each window across time. We plot the mean and standard deviation from five runs for each system. We see that using the micro-batch model in Spark has good relative adaptivity where the latency during failure is around 3x normal processing latency and that only 1 window is affected. Drizzle has similar behavior where the latency during failure increases from around 350ms to around 1s. Similar to Spark, Drizzle's latency also returns to normal after one window.

On the other hand Flink experiences severe slowdown during failure and the latency spikes to around 18s. Most of this slow down is due to the additional coordination required to stop and restart all the operators in the topology and to restore execution from the latest checkpoint. We also see that having such a huge delay means that the system is unable to catch up with the input stream and that it takes around 40s (or 4 windows) before latency returns to normal range.

4.4.5 Micro-batch Optimizations

As discussed in §4.2.6, one of the advantages of using the micro-batch model for streaming is that it enables additional optimizations to be used *within a batch*. In the case of the Yahoo streaming benchmark, as the output only requires the number of events in a window we can reduce the amount of data shuffled by aggregating counters for each window on the map side. We implemented this optimization in Drizzle and Spark by using the `reduceBy` operator instead of the `groupBy` operator and study the latency, throughput improvements bought about by this change.

Latency. Figure 4.7(a) shows the CDF of the processing latency when the optimization is enabled. Since Flink creates windows after the keys are partitioned, we could not directly apply the same optimization. In this case we see that using optimization leads to Drizzle getting around 94ms median latency and is 2x faster than Spark and 3x faster than Flink.

Throughput. Similarly we again measure the maximum throughput that can be achieved given a latency target in Figure 4.7(b). We see that using optimization within a batch can lead to significant wins in throughput as well. Drizzle is able to sustain around 35M events/second with a 100ms latency target, a target that both Spark and Flink are unable to meet for different reasons:

Spark due to scheduling overheads and Flink due to lack of batch optimizations. Similar to the previous comparison we see that the benefits from Drizzle become less pronounced at larger latency targets and that given a 1s target both Spark and Drizzle achieve similar throughput of 100M events/second. We use the optimized version for Drizzle and Spark in the following sections of the evaluation.

In summary, we find that by combining the batch-oriented data processing with the coarse grained scheduling in Drizzle we are able to achieve better performance than existing BSP systems like Spark and continuous operator systems like Flink. We also see that Drizzle also recovers faster from failures when compared to Flink and maintains low latency during recovery.

4.4.6 Adaptivity in Drizzle

We next evaluate the importance of group size in Drizzle and specifically how it affects adapativity in terms of fault tolerance and elasticity. Following that we show how our auto-tuning algorithm can find the optimal group size.

Fault tolerance with Group Scheduling

To measure the importance of group size for fault recovery in Drizzle, we use the same Yahoo workload as the previous section and we vary the group size. In this experiment we create checkpoints at the end of every group. We measure processing latency across windows and the median processing latency from five runs is shown in Figure 4.9(a).

We can see that using a larger group size can lead to higher latencies and longer recovery times. This is primarily because of two reasons. First, since we only create checkpoints at the end of every group having a larger group size means that more records would need to be replayed. Further, when machines fail pre-scheduled tasks need to be updated in order to reflect the new task locations and this process takes longer when there are larger number of tasks. In the future we plan to investigate if checkpoint intervals can be decoupled from group and better treatment of failures in pre-scheduling.

Handling Elasticity

To measure how Drizzle enables elasticity we consider a scenario where we start with the Yahoo Streaming benchmark but only use 64 machines in the cluster. We add 64 machines to the cluster after 4 minutes and measure how long it takes for the system to react and use the new machines. To measure elasticity we monitor the average latency to execute a micro-batch and results from varying the group size are shown in Figure 4.9(b).

We see that using a larger group size can delay the time taken to adapt to cluster changes. In this case, using a group size of 50 the latency drops from 150ms to 100ms within 10 seconds. On the other hand, using group size of 600 takes 100 seconds to react. Finally, as seen in the figure, elasticity can also lead to some performance degradation when the new machines are first used.

This is because some of the input data needs to be moved from machines that were being used to the new machines.

Group Size Tuning

To evaluate our group size tuning algorithm, we use the same Yahoo streaming benchmark but change the micro-batch size. Intuitively the scheduling overheads are inversely proportional to the micro-batch size, as small iterations imply there are more tasks to schedule. We run the experiment with the scheduling overhead target of 5% to 10% and start with a group size of 2. The progress of the tuning algorithm is shown in Figure 4.8(b) for micro-batch size of 100ms and 250ms.

We see that for a smaller micro-batch the overheads are high initially with the small group size and hence the algorithm increases the group size to 64. Following that as the overhead goes below 5% the group size is additively decreased to 49. In the case of the 250ms micro-batch we see that a group size of 8 is good enough to maintain a low scheduling overhead.

4.5 Drizzle Conclusion

In conclusion, this chapter shows that it is possible to achieve low latency execution for iterative workloads while using a BSP-style framework. Using the insight that we can decouple fine grained execution from coarse grained centralized scheduling, we build Drizzle, a framework for low latency iterative workloads. Drizzle reduces overheads by grouping multiple iterations and uses pre-scheduling and conflict-free shared variables to enable communication across iterations.

Chapter 5

Data-aware scheduling

Having looked at how we can optimize the control plane for machine learning workloads, we now turn our attention to the data plane. The efficient execution of I/O-intensive tasks is predicated on *data-aware scheduling*, i.e., minimizing the time taken by tasks to read their data. Widely deployed techniques for data-aware scheduling execute tasks on the same machine as their data (if the data is on one machine, as for input tasks) [10, 183] and avoid congested network links (when data is spread across machines, as for intermediate tasks) [11, 49]. However, despite these techniques, we see that production jobs in Facebook's Hadoop cluster are slower by 87% compared to *perfect data-aware scheduling* (§5.1.3). This is because, in multi-tenant clusters, compute slots that are ideal for data-aware task execution are often unavailable.

The importance of data-aware scheduling is increasing with rapid growth in data volumes [68]. To cope with this data growth and yet provide timely results, there is a trend of jobs using only a *subset* of their data. Examples include sampling-based approximate query processing systems [7, 15] and machine learning algorithms [28, 110]. A key property of such jobs is that they can compute on *any of the combinatorially many* subsets of the input dataset without compromising application correctness. For example, say a machine learning algorithm like stochastic gradient descent [28] needs to compute on a 5% uniform random sample of data. If the data is spread over 100 blocks then the scheduler can choose any 5 blocks and has $\binom{100}{5}$ input choices for this job.

Our goal is to *leverage the combinatorial choice of inputs for data-aware scheduling*. Current schedulers require the application to select a subset of the data on which the scheduler runs the job. This prevents the scheduler from taking advantage of available choices. In contrast, we argue for "late binding" *i.e.,* choosing the subset of data dynamically depending on the current state of the cluster (see Figure 5.1). This dramatically increases the number of data local slots for input tasks (e.g., map tasks), which increases the probability of achieving data locality even during high cluster utilizations.

In this chapter we describe the design and implementation of KMN, a scheduler that can leverage the choices available in sampling based workloads. We also describe how we can extend benefits of choice to intermediate stages and also explain how we can handle stragglers in upstream tasks using a *delay*-based approach.

5.1 Choices and Data-Awareness

In this section we first discuss application trends that result in increased choices for scheduling (§5.1.1). We then explain data-aware scheduling (§5.1.2) and quantify its potential benefit in production clusters (§5.1.3).

5.1.1 Application Trends

With the rapid increase in the volume of data collected, it has become prohibitively expensive for data analytics frameworks to operate on all of the data. To provide timely results, there is a trend towards trading off accuracy for performance. Quick results obtained from just part of the dataset are often *good enough*.

(1) Machine Learning: The last few years has seen the deployment of large-scale distributed machine learning algorithms for commercial applications like spam classification [104] and machine translation [32]. Recent advances [29] have introduced stochastic versions of these algorithms, for example stochastic gradient descent [28] or stochastic L-BFGS [145], that can use *small random data samples* and provide statistically valid results even for large datasets. These algorithms are iterative and each iteration processes only a small sample of the data. Stochastic algorithms are agnostic to the sample selected in each iteration and support flexible scheduling.

(2) Approximate Query Processing: Many analytics frameworks support approximate query processing (AQP) using standard SQL syntax (e.g., BlinkDB [7], Presto [63]). They power many popular applications like exploratory data analysis [34, 152] and interactive debugging [8]. For example, products analysts could use AQP systems to quickly decide if an advertising campaign needs to be changed based on a sample of click through rates. AQP systems can bound both the time taken and the quality of the result by selecting appropriately sized inputs (samples) to met the deadline and error bound. Sample sizes are typically small relative to the original data (often, one-twentieth to one-fifth [111]) and many equivalent samples exist. Thus, sample selection presents a significant opportunity for smart scheduling.

(3) Erasure Coded Storage: Rise in data volumes have also led to clusters employing efficient storage techniques like erasure codes [137]. Erasure codes provide fault tolerance by storing k extra parity blocks for every n data blocks. Using *any* n data blocks of the $(n + k)$ blocks, applications can compute their input. Such storage systems also provide choices for data-aware scheduling.

Note that while the above applications provide an opportunity to pick *any* subset of the input data, our system can also handle custom sampling functions, which generate samples based on application requirements.

5.1.2 Data-Aware Scheduling

Data aware scheduling is important for both the input as well as intermediate stages of jobs due to their IO-intensive nature. In the input stage, tasks reads their input from a single machine and the natural goal is *locality* i.e. to schedule the task on a machine that stores its input (§5.1.2).

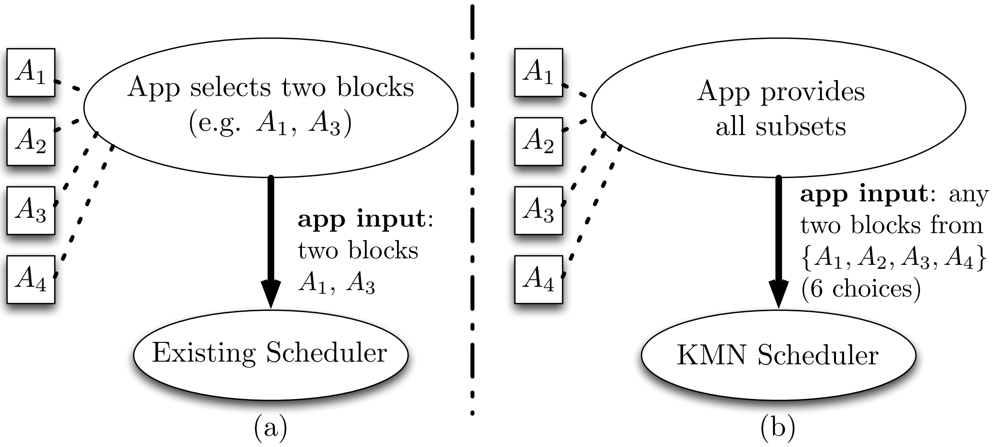

A_1
A_2
A_3
A_4
App selects two blocks (e.g. A_1, A_3)
app input: two blocks A_1, A_3
Existing Scheduler
(a)
A_1
A_2
A_3
A_4
App provides all subsets
app input: any two blocks from $\{A_1, A_2, A_3, A_4\}$ (6 choices)
KMN Scheduler
(b)

Figure 5.1: Late binding allows applications to specify more inputs than tasks and schedulers dynamically choose task inputs at execution time.

For intermediate stages, tasks have their input spread across multiple machines. In this case, it is not possible to co-locate the task with all its inputs. Instead, the goal in this case is to schedule the task at a machine that minimizes the time it takes to transfer all remote inputs. As over-subscribed cross-rack links are the main bottleneck in reads [47], we seek to *balance* the utilization of these links (§5.1.2).

Memory Locality for Input Tasks

Riding on the trend of falling memory prices, clusters are increasingly caching data in memory [14, 177]. As memory bandwidths are about $10x$ to $100x$ greater than the fastest network bandwidths, data reads from memory provide dramatic acceleration for the IO-intensive analytics jobs. However, to reap the benefits of in-memory caches, tasks have to be scheduled with *memory locality*, i.e., on the same machine that contains their input data. Obtaining memory locality is important for timely completion of interactive approximate queries [12]. Iterative machine learning algorithms typically run 100's of iterations and lack of memory locality results in huge slowdown per iteration and the overall job.

Achieving memory locality is a challenging problem in clusters. Since in-memory storage is used only as a cache, data stored in memory is typically not replicated. Further, the amount of memory in a cluster is relatively small (often by three orders of magnitude [12]) when compared to stable storage: this difference means that replicating data in memory is not practical. Therefore, techniques for improving locality [10] developed for disk-based replicated storage are insufficient; they rely on the probability of locality increasing with the number of replicas. Further, as job completion times are dictated by the slowest task in the job, improving performance requires memory locality for *all* its tasks [14].

These challenges are reflected in production Hadoop clusters. A Facebook trace from 2010 [10, 46] shows that less than 60% of tasks achieve locality *even with* three replicas. As in-memory data is not replicated, it is harder for jobs to achieve memory locality for all their tasks.

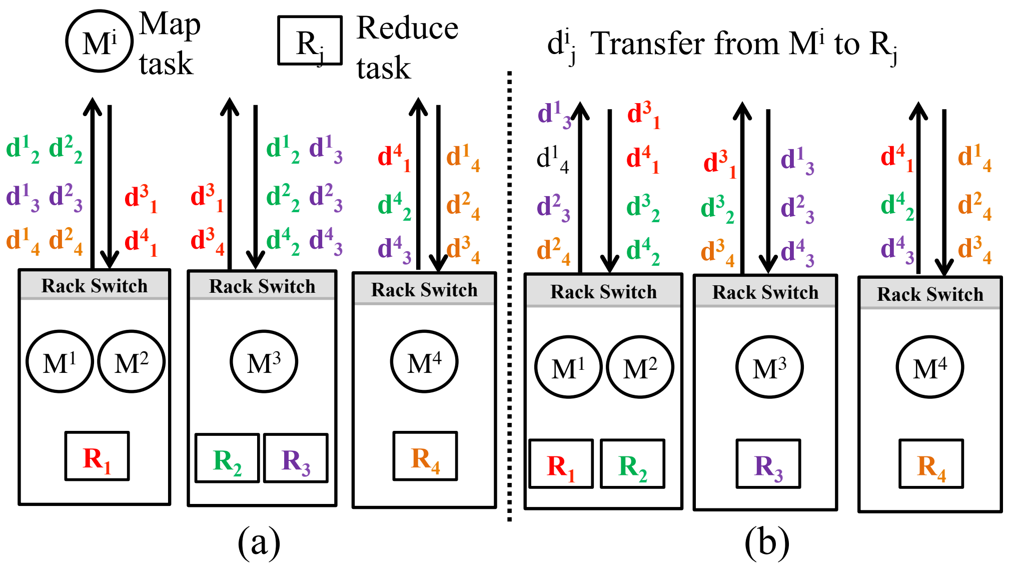


Figure 5.2: Value of balanced network usage for a job with 4 map tasks and 4 reduce tasks. The left-hand side has unbalanced cross-rack links (maximum of 6 transfers, minimum of 2) while the right-hand side has better balance (maximum of 4 transfers, minimum of 3).

Balanced Network for Intermediate Tasks

Intermediate stages of a job have communication patterns that result in their tasks reading inputs from many machines (e.g., all-to-all "shuffle" or many-to-one "join" stages). For I/O intensive intermediate tasks, the time to access data across the network dominates the running time, more so when intermediate outputs are stored in memory. Despite fast network links [165] and newer topologies [9,77], bandwidths between machines connected to the same rack switch are still $2\times$ to $5\times$ higher than to machines outside the rack switch via the network core. Thus the runtime for an intermediate stage is dictated by the amount of data transferred across racks. Prior work has also shown that reducing cross-rack hotspots, i.e., optimizing the *bottleneck* cross-rack link [11,49] can significantly improve performance.

Given the over-subscribed cross-rack links and the slowest tasks dictating job completion, it is important to *balance* traffic on the cross-rack links [26]. Figure 5.2 illustrates the result of having unbalanced cross-rack links. The schedule in Figure 5.2(b) results in a *cross-rack skew*, i.e., ratio of the highest to lowest used network links, of only $\frac{4}{3}$ (or 1.33) as opposed to $\frac{6}{2}$ (or 3) in Figure 5.2(a).

To highlight the importance of cross-rack skew, we used a trace of Hadoop jobs run in a Facebook cluster from 2010 [46] and computed the cross-rack skew ratio. Figure 5.3(a) shows a CDF of this ratio and is broken down by the number of map tasks in the job. From the figure we can see that for jobs with $50-150$ map tasks more than half of the jobs have a cross-rack skew of over $4\times$. For larger jobs we see that the median is $15\times$ and the 90^{th} percentile value is in excess of $30\times$.

5.1.3 Potential Benefits

How much do the above-mentioned lack of locality and imbalanced network usage hurt jobs? We estimate the potential for data-aware scheduling to speed up jobs using the same Facebook trace (described in detail in §5.5). We mimic job performance with an ideal data-aware scheduler using a "what-if" simulator. Our simulator is unconstrained and (i) assigns memory locality for *all* the tasks in the input phase (we assume $20\times$ speed up for memory locality [121] compared to reading data over the network based on our micro-benchmark) and (ii) places tasks to *perfectly balance* cross-rack links. We see that jobs speed up by 87.6% on average with such ideal data-aware scheduling.

Given these potential benefits, we have designed KMN, a scheduling framework that exploits the available choices to improve performance. At the heart of KMN lie scheduling techniques to increase locality for input (§5.2) stages and balance network usage for intermediate (§5.3) stages. In §5.4, we describe an interface that allows applications to specify all available choices to the scheduler.

5.2 Input Stage

For the input stage (*i.e.,* the map stage in MapReduce or the extract stage in Dryad) accounting for combinatorial choice leads to improved locality and hence reduced completion time. Here we analyze the improvements in locality in two scenarios: in §5.2.1 we look at jobs which can use any K of the N input blocks; in §5.2.2 we look at jobs which use a custom sampling function.

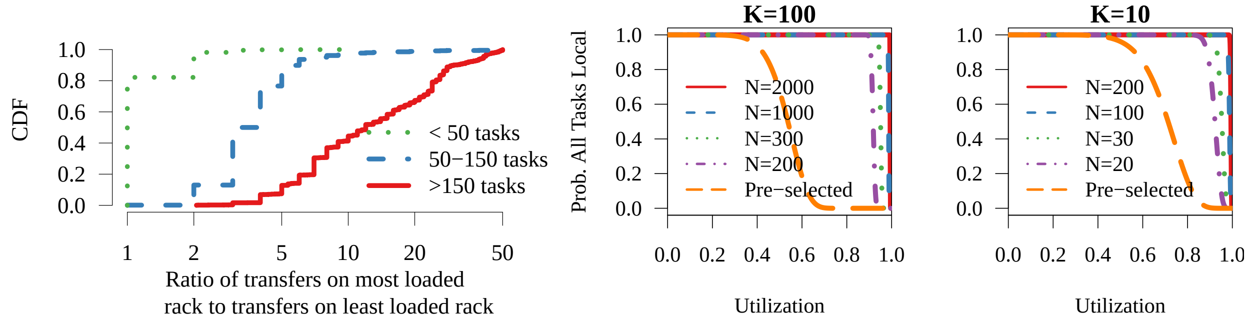
We assume a cluster with s compute slots per machine. Tasks operate on one input block each and input blocks are uniformly distributed across the cluster, this is in line with the block placement policy used by Hadoop. For ease of analysis we assume machines in the cluster are uniformly utilized (*i.e.,* there are no hot-spots). In our evaluation §5.5) we consider hot-spots due to skewed input-block and machine popularity.

5.2.1 Choosing any K out of N blocks

Many modern systems *e.g.,* BlinkDB [7], Presto [63], AQUA [4] operate by choosing a random subset of blocks from shuffled input data. These systems rely on the observation that block sampling [43] is statistically equivalent to uniform random sampling (page 243 in [159]) when each block is itself a random sample of the overall population. Given a sample size K, these systems can operate on any K input blocks *i.e.,* for an input of size N the scheduler can choose any one of $\binom{N}{K}$ combinations.

In the cluster setup described above, the probability that a task operating on an input block gets locality is $p_t = 1 - u^s$ where u is the cluster utilization (probability that all slots in a machine are busy is $= u^s$). For such a cluster the probability for K out of N tasks getting locality is given by the binomial CDF function with the probability of success $= p_t$, i.e., $1 - \sum_{i=0}^{K-1} \binom{N}{i} p_t^i (1-p_t)^{N-i}$.

The dominant factor in this probability is the ratio between K and N. In Figure 5.3(b) we fix the number of slots per machine to $s = 8$ and plot the probability of $K = 10$ and $K = 100$



(a) CDF of cross-rack skew for the Facebook trace split by number of map tasks. Reducing cross-rack skew improves intermediate stage performance.

(b) Probability of input-stage locality when choosing any K out of N blocks. The scheduler can choose to execute a job on any of $\binom{N}{K}$ samples.

Figure 5.3: Cross-rack skew and input-stage locality simulation.

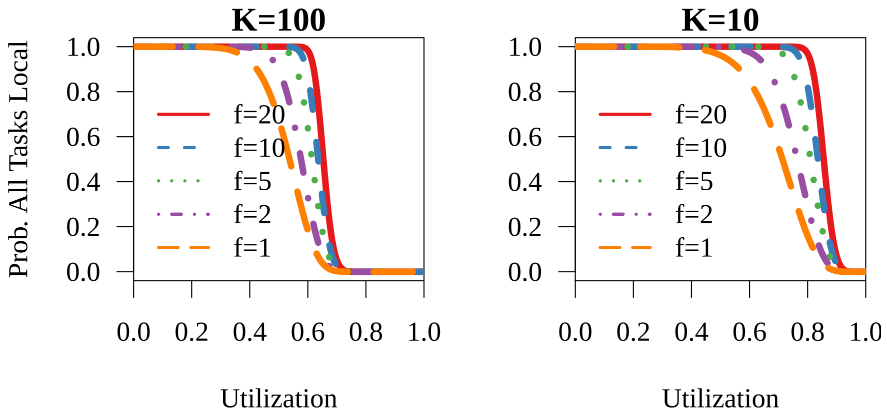


Figure 5.4: Probability of input-stage locality when using a sampling function which outputs f disjoint samples. Sampling functions specify additional constraints for samples.

tasks getting locality in a job with varying input size N and varying cluster utilization. We observe that the probability of achieving locality is high even when 90% of the cluster is utilized. We also compare this to a baseline that does not exploit this combinatorial choice and pre-selects a random K blocks *beforehand*. For the baseline the probability that all tasks are local drops dramatically even with cluster utilization of 60% or less.

5.2.2 Custom Sampling Functions

Some systems require additional constraints on the samples used and use custom sampling functions. These sampling functions can be used to produce several K-block samples and the scheduler can pick *any* sample. The scheduler is however constrained to use *all* of the K-blocks from one sample. We consider a sampling function that produces f disjoint samples and analyze locality improvements in this setting.

As noted previous, the probability of a task getting locality is $p_t = 1 - u^s$. The probability that all K blocks in a sample get locality is p_t^K. Since the f samples are disjoint (and therefore the probability of achieving locality is independent) the probability that at least one among the f samples can achieve locality is $p_j = 1 - (1 - p_t^K)^f$. Figure 5.4 shows the probability of $K = 10$ and $K = 100$ tasks achieving locality with varying utilization and number of samples. We see that the probability of achieving locality significantly increases with f. At $f = 5$ we see that small jobs (10 tasks) can achieve complete locality even when the cluster is 80% utilized.

We thus find that accounting for combinatorial choices can greatly improve locality for the input stage. Next we analyze improvements for intermediate stages.

5.3 Intermediate Stages

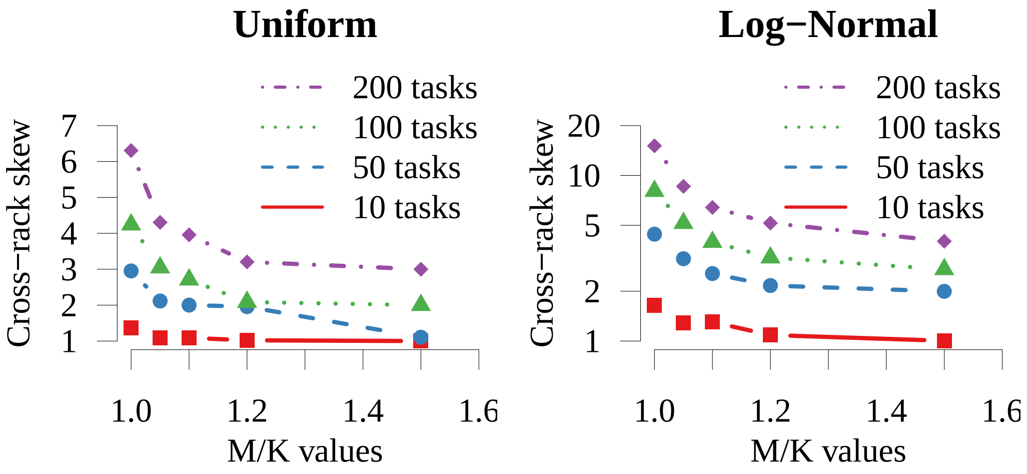
Intermediate stages of jobs commonly involve one-to-all (broadcast), many-to-one (coalesce) or many-to-many (shuffle) network transfers [48]. These transfers are network-bound and hence, often slowed down by congested cross-rack network links. As described in §5.1.2, data-aware scheduling can improve performance by better placement of both upstream and downstream tasks to balance the usage of cross-rack network links.

While effective heuristics can be used in scheduling downstream tasks to balance network usage (we deal with this in §5.4), they are nonetheless limited by the locations of the outputs of upstream tasks. Scheduling upstream tasks to balance the locations of their outputs across racks is often complicated due to many dynamic factors in clusters. First, they are constrained by data locality (§5.2) and compromising locality is detrimental. Second, the utilization of the cross-rack links when downstream tasks start executing are hard to predict in multi-tenant clusters. Finally, even the size of upstream outputs varies across jobs and are not known beforehand.

We overcome these challenges by scheduling a *few additional* upstream tasks. For an upstream stage with K tasks, we schedule M tasks ($M > K$). Additional tasks increase the likelihood that task outputs are distributed across racks. This allows us to choose the "best" K out of M upstream tasks, out of $\binom{M}{K}$ choices, to minimize cross-rack network utilization. In the rest of this section, we show analytically that a few additional upstream tasks can significantly reduce the imbalance (§5.3.1). §5.3.2 describes a heuristic to pick the best K out of M upstream tasks. However, not all M upstream tasks may finish simultaneously because of stragglers; we modify our heuristic to account for stragglers in §5.3.3.

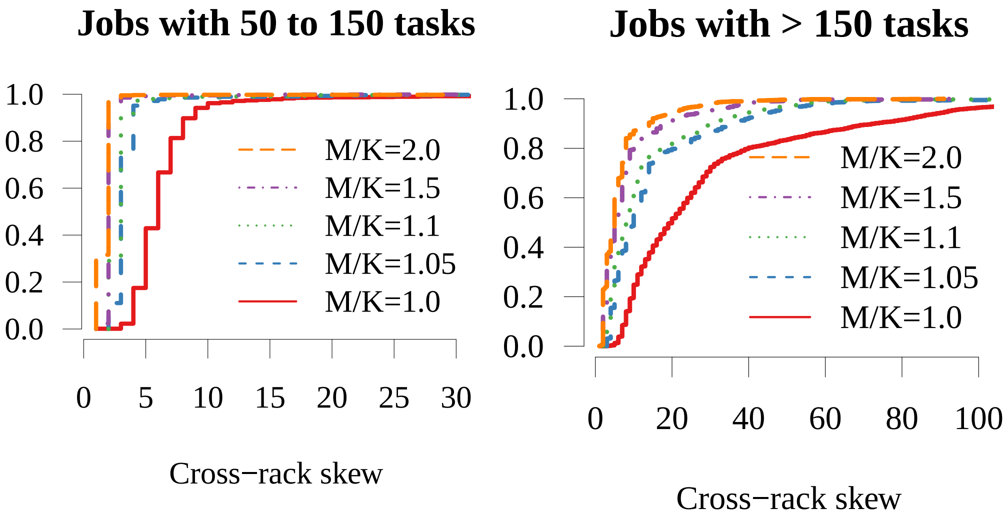
5.3.1 Additional Upstream Tasks

While running additional tasks can balance network usage, it is important to consider how many additional tasks are required. Too many additional tasks can often lead to worsening of overall cluster performance.



(a) (b)

Figure 5.5: Cross-rack skew as we vary M/K for uniform and log-normal distributions. Even 20% extra upstream tasks greatly reduces network imbalance for later stages.



(a) (b)

Figure 5.6: CDF of cross-rack skew as we vary M/K for the Facebook trace.

We analyze this using a simple model of the scheduling of upstream tasks. For simplicity, we assume that upstream task outputs are equal in size and network links are equally utilized. We only model tasks at the level of racks and evaluate the cross-rack skew (ratio of the rack with largest and smallest number of upstream tasks) using both synthetic distributions of upstream task locations as well as data from our Facebook trace.

Synthetic Distributions: We first consider a scheduler that places tasks on racks uniformly at random. Figure 5.5(a) plots the cross-rack skew in a 100 rack cluster for varying values of K (*i.e.,* the stage's desired number of tasks) and M/K (*i.e.,* the fraction of additional tasks launched). We can see that even with a scheduler that places the upstream tasks uniformly, there is significant skew for large jobs when there are no additional tasks ($\frac{M}{K} = 1$). This is explained by the balls and bins problem [126] where the maximum imbalance is expected to be $O(logn)$ when distributing n balls.

However, we see that even with 10% to 20% additional tasks ($\frac{M}{K} = 1.1 - 1.2$) the cross-rack skew is reduced by $\geq 2\times$. This is because when the number of upstream tasks, n is > 12, $0.2n > logn$. Thus, we can avoid most of the skew with just a few extra tasks.

We also repeat this study with a log normal distribution ($\theta = 0, m = 1$) of upstream task placement; this is more skewed compared to the uniform distribution. However, even with a log-normal distribution, we again see that a few extra tasks can be very effective at reducing skew. This is because the expected value of the most loaded bin is still linear and using $0.2n$

additional tasks is sufficient to avoid most of the skew.

Facebook Distributions: We repeat the above analysis using the number and location of upstream tasks of a phase in the Facebook trace (used in §5.1.2). Recall the high cross-rack skew in the Facebook trace. Despite that, again, a few additional tasks suffices to eliminate a large fraction of the skews. Figure 5.6 plots the results for varying values of $\frac{M}{K}$ for different jobs. A large fraction of the skew is reduced by running just 10% more tasks. This is nearly 66% of the reductions we get using $\frac{M}{K} = 2$.

In summary we see that running a few extra tasks is an effective strategy to reduce skew, both with synthetic as well as real-world distributions. We next look at mechanisms that can help us achieve such reduction.

5.3.2 Selecting Best Upstream Outputs

The problem of selecting the best K outputs from the M upstream tasks can be stated as follows: We are given M upstream tasks $U = u_1 ... u_M$, R downstream tasks $D = d_1 ... d_R$ and their corresponding rack locations. Let us assume that tasks are distributed over racks $1...L$ and let $U' \subset U$ be some set of K upstream outputs. Then for each rack we can define the uplink cost C_{2i-1} and downlink cost C_{2i} using a cost function $C_i(U', D)$. Our objective then is to select U' to minimize the most loaded link i.e.

$$\arg\min_{U'} \max_{i \in 2L} C_i(U', D)$$

While this problem is NP-Hard [174], many approximation heuristics have been developed. We use a heuristic that corresponds to spreading our choice of K outputs across as many racks as possible.[1]

[1] This problem is an instance of the facility location problem [53] where we have a set of clients (downstream

Algorithm 3 Choosing K upstream outputs out of M using a round-robin strategy

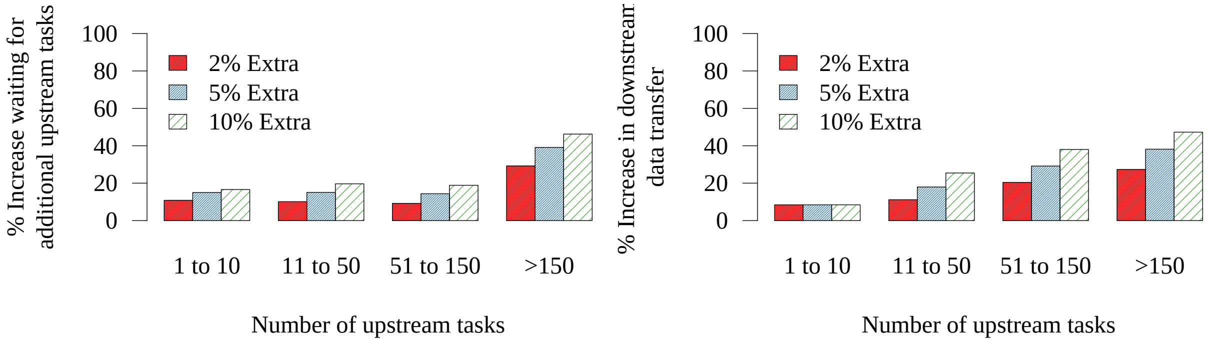
```
Given: upstreamTasks - list with $rack, index$ within rack for each task
Given: $K$ - number of tasks to pick
// Number of upstream tasks in each rack
upstreamRacksCount = map()
// Initialize
for $task$ in upstreamTasks do
    upstreamRacksCount[$task.rack$] += 1
// Sort the tasks in round-robin fashion
roundRobin = upstreamTasks.sort(CompareTasks)
chosenK = roundRobin[0 : $K$]
return chosenK
Function{CompareTasks}{$task1, task2$}
    if $task1.idx$ != $task2.idx$ then
        // Sort first by index
        return $task1.idx < task2.idx$
    else
        // Then by number of outputs
        numRack1 = upstreamRacksCount[$task1.rack$]
        numRack2 = upstreamRacksCount[$task2.rack$]
        return numRack1 $>$ numRack2
EndFunction
```

Our implementation for this approximation heuristic is shown in Algorithm 3. We start with the list of upstream tasks and build a hash map that stores how many tasks were run on each rack. Next we sort the tasks first by their index within a rack and then by the number of tasks in the rack. This sorting criteria ensures that we first see one task from each rack, thus ensuring we spread our choices across racks. We use an additional heuristic of favoring racks with more outputs to help our downstream task placement techniques (§5.4.2). The main computation cost in this method is the sorting step and hence this runs in $O(MlogM)$ time for M tasks.

5.3.3 Handling Upstream Stragglers

While the previous section described a heuristic to pick the best K out of M upstream outputs, waiting for all M can be inefficient due to *stragglers*. Stragglers in the upstream stage can delay completion of some tasks which cuts into the gains obtained by balancing the network links. Stragglers are a common occurrence in clusters with many clusters reporting significantly slow tasks despite many prevention and speculation solutions [11, 13, 186]. This presents a trade-off in waiting for all M tasks and obtaining the benefits of choice in picking upstream outputs against the wasted time for completion of all M upstream tasks including stragglers. Our solution for

tasks), set of potential facility locations (upstream tasks), a cost function that maps facility locations to clients (link usage). Our heuristic follows from picking a facility that is farthest from the existing set of facilities [65].



(a) Percentage of time spent waiting for additional upstream tasks when running 2%, 5% or 10% extra tasks. Stage completion time can be increased by up $20\% - 40\%$ due to stragglers.

(b) Percentage of additional time spent in downstream data transfer when not using choices from 2%, 5% or 10% extra tasks. Decrease in choice increases data transfer time by $20\% - 40\%$.

Figure 5.7: Simulations to show how choice affects stragglers and downstream transfer

this problem is to schedule downstream tasks at some point after K upstream tasks have completed but not wait for the stragglers in the M tasks. We quantify this trade-off with analysis and micro-benchmarks.

Stragglers vs. Choice

We study the impact of stragglers in the Facebook trace when we run 2%, 5% and 10% extra tasks (i.e., $\frac{M}{K} = 1.02, 1.05, 1.1$). We compute the difference between the time taken for the fastest K tasks and the time to complete all M tasks. Figure 5.7(a) shows that waiting for the extra tasks can inflate the completion of the upstream phase by $20\% - 40\%$ (for jobs with > 150 tasks). Also, the trend of using a large number of small tasks [135] for interactive jobs will only worsen such inflation. On the other hand avoiding upstream stragglers by using the fastest tasks reduces the available choice. Consequently, the time taken for downstream data transfer increases. The lack of choices from extra tasks means we cannot balance network usage. Figure 5.7(b) shows that not using choice from additional tasks can increase data transfer time by 20% for small jobs (11 to 50 tasks) and up to 40% for large jobs (> 150 tasks). We now devise a simple approach to balance between the above two factors—waiting for upstream stragglers versus losing choice for downstream data transfer.

Delayed Stage Launch

The problem we need to solve can be formulated as: we have M upstream tasks $u_1, u_2, ..., u_M$ and for each task we have corresponding rack locations. Our goal is to find the optimal *delay* after the first K tasks have finished, such that the overall time taken is minimized. In other words, our goal is to find the optimal K' tasks to wait for before starting the downstream tasks.

We begin with assuming an oracle that can give us the task finish times for all the tasks. Given such a oracle we can sort the tasks in an increasing order of finish times such that $F_j \geq F_i \, \forall j > i$. Let us define the waiting delay for tasks $K + 1$ to M as $D_i = F_i - F_k \, \forall i > k$. We also assume that given K' tasks, we can compute the optimal K tasks to use (§5.3.2) and the estimated transfer time $S_{K'}$.

Our problem is to pick K' ($K \leq K' \leq M$) such that the total time for the data transfer is minimized. That is we need to pick K' such that $F_k + D_{k'} + S_{k'}$ is minimized. In this equation F_k is known and independent of K'. Of the other two, $D_{k'}$ increases as k' goes from K to M, while $S_{k'}$ decreases. However as the sum of an increasing and decreasing function is not necessarily convex [2] it isn't easy to minimize the total time taken.

Delay Heuristic: While the brute-force approach would require us to try all values from K to M, we develop two heuristics that allow us to bound the search space and quickly find the optimal value of K'.

- *Bounding transfer*: At the beginning of the search procedure we find the maximum possible improvement we can get from picking the best set of tasks. Whenever the delay $D_{K'}$ is greater than the maximum improvement, we can stop the search as the succeeding delays will increase the total time.
- *Coalescing tasks*: We can also coalesce a number of task finish events to further reduce the search space. For example we can coalesce task finish events which occur close together by time i.e., cases $D_{i+1} - Di < \delta$. This will mean our result is off by at most δ from the optimal, but for small values of δ we can coalesce tasks of a wave that finish close to each other.

Using these heuristics we can find the optimal number of tasks to wait for quickly. For example, in the Facebook trace described before using $M/K = 1.1$ or 10% extra tasks, determining the optimal wait time for a job requires looking at less than 4% of all configurations when we use a coalescing error of 1%. We found coalescing tasks to be particularly useful as even with a δ of 0.1% we need to look at around 8% of all possible configurations. Running without any coalescing is infeasible since it takes ≈ 1000 ms.

Finally, we relax our assumption of an oracle as follows. While the task finish times are not exactly known beforehand, we use job sizes to figure out if the same job has been run before. Based on this we use the job history to predict the task finish times. This approach should work well for clusters that have many jobs run periodically [83]. In case the job history is not available we can fit the tasks length distribution using the first few task finish times and use that to get approximate task finish times for the rest of the tasks [60].

5.4 KMN Implementation

We have built KMN on top of Apache Spark [184] version 0.7.3 and KMN consists of 1400 lines of Scala code. We next discuss some of the features and challenges in our implementation.

[2] Take any non-convex function and make its increasing region F_i and its decreasing region F_d and it can be seen that the sum isn't convex.

5.4.1 Application Interface

We define a `blockSample` operator which jobs can use to specify input constraints (for instance, use K blocks from file F) to the framework.The `blockSample` operator takes two arguments: the ratio $\frac{K}{N}$ and a *sampling function* that can be used to impose constraints. The sampling function can be used to choose user-defined sampling algorithms (*e.g.,* stratified sampling). By default the sampling function picks any K of N blocks.

Consider an example SQL query and its corresponding Spark [184] version shown in Figure 5.8. To run the same query in KMN we just need to prefix the query with the `blockSample` operator. The *sampler* argument is a Scala closure and passing `None` causes the scheduler to use the default function which picks any K out of the N input blocks. This design can be readily adapted to other systems like Hadoop MapReduce and Dryad.

KMN also provides an interface for jobs to introspect which samples where used in a computation. This can be used for error estimation using algorithms like Bootstrap [6] and also provides support for queries to be repeated. We implement this in KMN by storing the K partitions used during computation as a part of a job's lineage. Using the lineage also ensures that the same samples are used if the job is re-executed during fault recovery [184].

5.4.2 Task Scheduling

We modify Spark's scheduler in KMN to implement the techniques described in earlier sections.

Input Stage

Schedulers for frameworks like MapReduce or Spark typically use a slot-based model where the scheduler is invoked whenever a slot becomes available in the cluster. In KMN, to choose any K out of N blocks we modify the scheduler to run tasks on blocks local to the first K available slots. To ensure that tasks don't suffer from resource starvation while waiting for locality, we use a timeout after which tasks are scheduled on any available slot. Note that, choosing the first K slots provides a sample similar or slightly better in quality compared to existing systems like Aqua [4] or BlinkDB [7] that reuse samples for short time periods. To schedule jobs with custom sampling functions, we similarly modify the scheduler to choose among the available samples and run the computation on the sample that has the highest locality.

Intermediate Stage

Existing cluster computing frameworks like Spark and Hadoop place intermediate stages without accounting for their dependencies. However smarter placement which accounts for a tasks' dependencies can improve performance. We implemented two strategies in KMN:

Greedy assignment: The number of cross-rack transfers in the intermediate stage can be reduced by co-locating map and reduce tasks (more generally any dependent tasks). In the greedy placement strategy we maximize the number of reduce tasks placed in the rack with the most map

```
// SQL Query
SELECT status, SUM(quantity)
FROM items
GROUP BY status

// Spark Query
kv = file.map{ li =>
  (li.l_linestatus,li.quantity)}
result = kv.reduceByKey{(a,b) =>
  a + b}.collect()

// KMN Query
sample = file.blockSample(0.1, sampler=None)
kv = sample.map{ li =>
  (li.l_linestatus,li.quantity)}
result = kv.reduceByKey{(a,b) =>
  a + b}.collect()
```

Figure 5.8: An example of a query in SQL, Spark and KMN

tasks. This strategy works well for small jobs where network usage can be minimized by placing all the reduce tasks in the same rack.

Round-robin assignment: While greedy placement minimizes the number of transfers from map tasks to reduce tasks it results in most of the data being sent to one or a few racks. Thus the links into these racks are likely to be congested. This problem can be solved by distributing tasks across racks while simultaneously minimizing the amount of data sent across racks. This can be achieved by evenly distributing the reducers across racks with map tasks. This strategy can be shown to be optimal if we know the map task locations and is similar in nature to the algorithm described in §5.3.2. We perform a more detailed comparison of the two approaches in §5.5

5.4.3 Support for extra tasks

One consequence of launching extra tasks to improve performance is that the cluster utilization could be affected by these extra tasks. To avoid utilization spikes, in KMN the value for M/K (the percentage of extra tasks to launch) can only be set by the cluster administrator and not directly by the application. Further, we implemented support for killing tasks once the scheduler decides that the tasks' output is not required. Killing tasks in Spark is challenging as tasks are run in threads and many tasks share the same process. To avoid expensive clean up associated with killing threads [96], we modified tasks in Spark to periodically poll and check a status bit. This means that tasks sometimes could take a few seconds more before they are terminated, but we found that this overhead was negligible in practice.

In KMN, using extra tasks is crucial in extending the flexibility of many choices throughout the DAG. In §5.2 and §5.3 we discussed how to use the available choices in the input and intermediate

Job Size	% of Jobs
1 to 10	57.23
11 to 100	41.02
> 100	4.82

Table 5.1: Distribution of job sizes in the scaled down version of the Facebook trace used for evaluation.

stages in a DAG. However, jobs created using frameworks like Spark or DryadLINQ can extend across many more stages. For example, complex SQL queries may use a map followed a shuffle to do a group-by operation and follow that up with a join. One solution to this would be run more tasks than required in every stage to retain the ability to choose among inputs in succeeding stages. However we found that in practice this does not help very much. In frameworks like Spark which use lazy evaluation, every stage following than the first stage is treated as an intermediate stage. As we use a round-robin strategy to schedule intermediate tasks (§5.4.2), the outputs from the first intermediate stage are already well spread out across the racks. Thus there isn't much skew across racks that affects the performance of following stages. In evaluation runs we saw no benefits for later stages of long DAGs.

5.5 KMN Evaluation

In addition to machine learning workload, we evaluate the benefits of KMN using two approaches: first we run approximate queries used in production at Conviva, a video analytics company, and study how KMN compares to using existing schedulers with pre-selected samples. Next we analyze how KMN behaves in a shared cluster, by replaying a workload trace obtained from Facebook's production Hadoop cluster.

Metric: In our evaluation we measure percentage improvement of job completion time when using KMN. We define percentage improvement as:

$$\% \text{ Improvement} = \frac{\text{Baseline Time} - \text{KMN Time}}{\text{Baseline Time}} \times 100$$

Our evaluation shows that,

- KMN improves real-world sampling-based queries from Conviva by more than 50% on average across various sample sizes and machine learning workloads by up to 43%.
- When replaying the Facebook trace, on an EC2 cluster, KMN can improve job completion time by 81% on average (92% for small jobs)
- By using 5% – 10% extra tasks we can balance bottleneck link usage and decrease shuffle times by 61% – 65% even for jobs with high cross-rack skew.

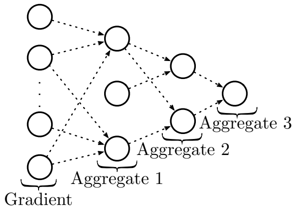
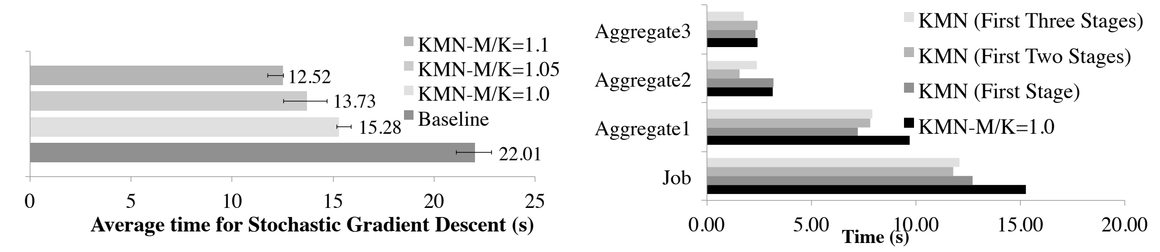


Figure 5.9: Execution DAG for Stochastic Gradient Descent (SGD).

Job Size	% Overall	% Map Stage	% Shuffle
1 to 10	92.8	95.5	84.61
11 to 100	78	94.1	28.63
> 100	60.8	95.4	31.02

Table 5.2: Improvements over baseline, by job size and stage



(a) Overall improvement when running Stochastic Gradient Descent using KMN

(b) Breakdown of aggregation times when using KMN for different number of stages in SGD

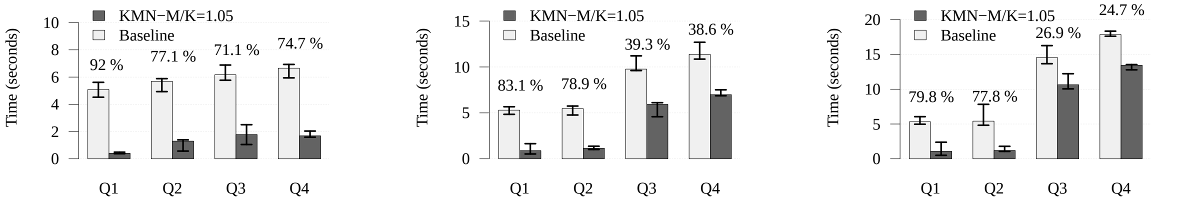
Figure 5.10: Benefits from using KMN for Stochastic Gradient Descent

5.5.1 Setup

Cluster Setup:We run all our experiments using 100 m2.4xlarge machines on Amazon's EC2 cluster, with each machine having 8 cores, 68GB of memory and 2 local drives. We configure Spark to use 4 slots and 60 GB per machine. To study memory locality we cache the input dataset before starting each experiment. We compare KMN with a baseline that operates on a pre-selected sample of size K and does not employ any of the shuffle improvement techniques described in §5.3, §5.4. We also label the fraction of extra tasks run (*i.e.,* M/K), so KMN-$M/K = 1.0$ has $K = M$ and KMN-$M/K = 1.05$ has 5% extra tasks. Finally, all experiments were run at least three times and we plot median values across runs and use error bars to show minimum and maximum values.
Workload: Our evaluation uses a workload trace from Facebook's Hadoop cluster [46]. The traces are from a mix of interactive and batch jobs and capture over half a million jobs on a 3500 node cluster. We use a scaled down version of the trace to fit within our cluster and use the same inter-arrival times and the task-to-rack mapping as in the trace. Unless specified, we use 10% sampling when running KMN for all jobs in the trace.

5.5.2 Benefits of KMN

We evaluate the benefits of using KMN on three workloads: real-world approximate queries from Conviva, a machine learning workload running stochastic gradient descent and a Hadoop workload trace from Facebook.



(a) Conviva queries with 1% Sampling (b) Conviva queries with 5% Sampling (c) Conviva queries with 10% Sampling

Figure 5.11: Comparing baseline and KMN-1.05 with sampling-queries from Conviva. Numbers on the bars represent percentage improvement when using KMN-$M/K = 1.05$.

Conviva Sampling jobs

We first present results from running 4 real-world sampling queries obtained from Conviva, a video analytics company. The queries were run on access logs obtained across a 5-day interval. We treat the entire data set as N blocks and vary the sampling fraction (K/N) to be 1%, 5% and 10%. We run the queries at 50% cluster utilization and run each query multiple times.

Figure 5.11 shows the median time taken for each query and we compare KMN-$M/K = 1.05$ to the baseline that uses pre-selected samples. For query 1 and query 2 we can see that KMN gives 77%–91% win across 1%, 5% and 10% samples. Both these queries calculate summary statistics across a time window and most of the computation is performed in the map stage. For these queries KMN ensures that we get memory locality and this results in significant improvements. For queries 3 and 4, we see around 70% improvement for 1% samples, and this reduces to around 25% for 10% sampling. Both these queries compute the number of distinct users that match a specified criteria. While input locality also improves these queries, for larger samples the reduce tasks are CPU bound (while they aggregate values).

Machine learning workload

Next, we look at performance benefits for a machine learning workload that uses sampling. For our analysis, we use Stochastic Gradient Descent (SGD). SGD is an iterative method that scales to large datasets and is widely used in applications such as machine translation and image classification. We run SGD on a dataset containing 2 million data items, where each each item contains 4000 features. The complete dataset is around 64GB in size and each of our iterations operates on a 1% sample 1% of the data. Thus the random sampling step reduces the cost of gradient computation by $100\times$ but maintains rapid learning rates [143]. We run 10 iterations in each setting to measure the total time taken for SGD.

Each iteration consists of a DAG comprised of a map stage where the gradient is computed on sampled data items and the gradient is then aggregated from all points. The aggregation step can be efficiently performed by using an aggregation tree as shown in Figure 5.9. We implement

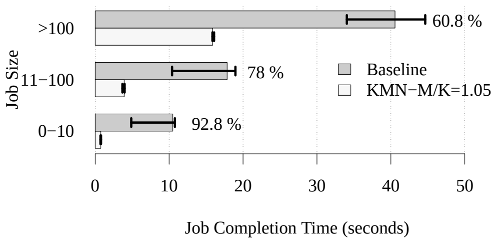


Figure 5.12: Overall improvement from KMN compared to baseline. Numbers on the bar represent percentage improvement using KMN-$M/K = 1.05$.

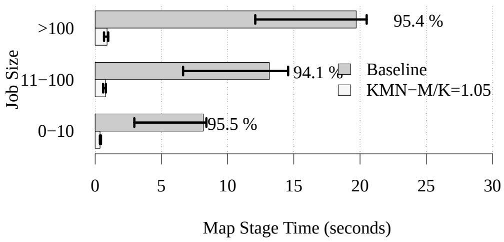


Figure 5.13: Improvement due to memory locality for the Map Stage for the Facebook trace. Numbers on the bar represent percentage improvement using KMN-$M/K = 1.05$.

the aggregation tree using a set of shuffle stages and use KMN to run extra tasks at each of these aggregation stages.

The overall benefits from using KMN are shown in Figure 5.10(a). We see that KMN-$M/K = 1.1$ improves performance by 43% as compared to the baseline. These improvements come from a combination of improving memory locality for the first stage and by improving shuffle performance for the aggregation stages. We further break down the improvements by studying the effects of KMN at every stage in Figure 5.10(b).

When running extra tasks for only the first stage (gradient stage), we see improvements of around 26% for the first aggregation (Aggregate 1); see KMN (First Stage). Without extra tasks the next two aggregation stages (Aggregate 2 and Aggregate 3) behave similar to KMN-$M/K = 1.0$. When extra tasks are spawned for later stages too, benefits propagate and we see 50% improvement in the second aggregation (Aggregate-2) while using KMN for the first two stages. However, propagating choice across stages does impose some overheads. Thus even though we see that KMN (First Three Stages) improves the performance of the last aggregation stage (Aggregate 3), running extra tasks slows down the overall job completion time (Job). This is because the final aggregation steps usually have fewer tasks with smaller amounts of data, which makes running extra tasks not worth the overhead. We plan to investigate techniques to estimate this trade-off and automatically determine which stages to use KMN for in the future.

Facebook workload

We next quantify the overall improvements across the trace from using KMN. To do this, we use a baseline configuration that mimics task locality from the original trace while using pre-selected samples. We compare this to KMN-$M/K = 1.05$ that uses 5% extra tasks and a round-robin reducer placement strategy (§5.4.2). The results showing average job completion time broken down by job size is shown in Figure 5.12 and relative improvements are shown in Table 5.2. As seen in the figure, using KMN leads to around 92% improvement for small jobs with < 10 tasks and more than 60% improvement for all other jobs. Across all jobs KMN-$M/K = 1.05$ improves performance by 81%, which is 93% of the potential win (§5.1.3).

To quantify where we get improvements from, we break down the time taken by different stages

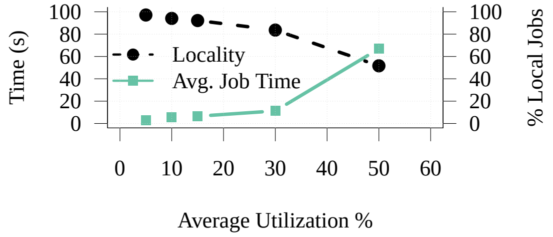


Figure 5.14: Job completion time and locality as we increase utilization.

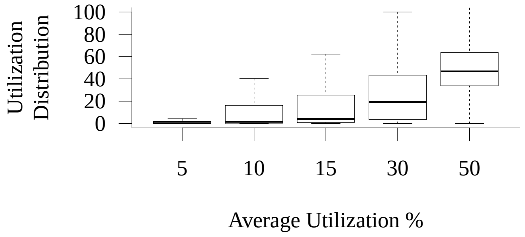


Figure 5.15: Boxplot showing utilization distribution for different values of average utilization.

of a job. Improvements for the input stage or the map stage are shown in Figure 5.13. We can see that using KMN we are able to get memory locality for almost all the jobs and this results in around 94% improvement in the time taken for the map stage. This is consistent with the predictions from our model in §5.2 and shows that pushing down sampling to the run-time can give tremendous benefits. The improvements in the shuffle stage are shown in Figure 5.16. For small jobs with < 10 tasks we get around 85% improvement and these are primarily because we co-locate the mappers and reducers for small jobs and thus avoid network transfer overheads. For large jobs with > 100 tasks we see around 30% improvement due to reduction in cross-rack skew.

5.5.3 Input Stage Locality

Next, we attempt to measure how the locality obtained by KMN changes with cluster utilization. As we vary the cluster utilization, we measure the average job completion time and fraction of jobs where all tasks get locality. The results shown in Figure 5.14 show that for up to 30% average utilization, KMN ensures that more than 80% of jobs get perfect locality. We also observed significant variance in the utilization during the trace replay and the distribution of utilization values is shown as a boxplot in Figure 5.15. From this figure we can see that while average utilization is 30% we observe utilization spikes of up to 90%. Because of such utilization spikes, we see periods of time where all jobs do not get locality.

Finally, at 50% average utilization (utilization spikes $> 90\%$) only around 45% of jobs get locality. This is lower than predictions from our model in §5.2. There are two reasons for this difference: First, our experimental cluster has only 400 slots and as we do 10% sampling ($K/N = 0.1$), the setup doesn't have enough choices for jobs with > 40 map tasks. Further the utilization spikes also are not taken into account by the model and jobs which arrive during a spike do not get locality.

5.5.4 Intermediate Stage Scheduling

In this section we evaluate scheduling decisions by KMN for intermediate stages. First we look at the benefits from running additional map tasks and then evaluate the delay heuristic used for straggler mitigation. Finally we also measure KMN's sensitivity to reducer placement strategies.

Job Size	$M/K = 1.0$	M/K =1.05	M/K =1.1
1 to 10	85.04	84.61	83.76
11 to 100	27.5	28.63	28.18
$>$ 100	14.44	31.02	36.35

Table 5.3: Shuffle time improvements over baseline while varying M/K

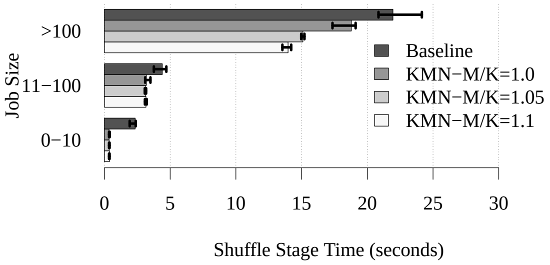


Figure 5.16: Shuffle improvements when running extra tasks.

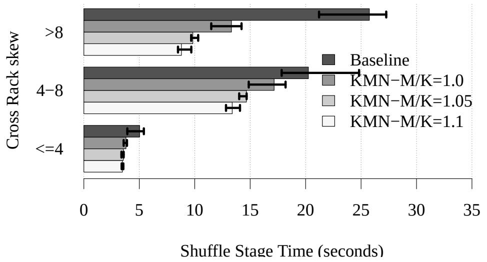


Figure 5.17: Difference in shuffle performance as cross-rack skew increases

Effect of varying M/K

We evaluate the effect of running extra map tasks (i.e $M/K > 1.0$) and measure how that influences the time taken for shuffle operations. For this experiment we wait until all the map tasks have finished and then calculate the best reducer placement and choose the best K map outputs as per techniques described in §5.3.2. The average time for the shuffle stage for different job sizes is shown in Figure 5.16 and the improvements with respect to the baseline are shown in Table 5.3. From the figure, we see that for small jobs with less than 10 tasks there is almost no improvement from running extra tasks as they usually do not suffer from cross-rack skew. However for large jobs with more than 100 tasks, we now get up to 36% improvement in shuffle time over the baseline.

Further, we can also analyze how the benefits are sensitive to the cross-rack skew. We plot the average shuffle time split by cross-rack skew in Figure 5.17. Correspondingly we list the improvements over the baseline in Table 5.4. We can see that for jobs which have low cross-rack skew, we get up to 33% improvement when using KMN-$M/K = 1.1$. Further, for jobs which have cross-rack skew > 8, we get up to 65% improvement in shuffle times and a 17% improvement over $M/K = 1$.

Delayed stage launch

We next study the impact of stragglers and the effect of using the delayed stage launch heuristic from §5.3.3. We run the Facebook workload at 30% cluster utilization with KMN-$M/K = 1.1$ and compare our heuristic to two baseline strategies. In one case we wait for the first K map tasks to finish before starting the shuffle while in the other case we wait for all M tasks for finish. The performance break down for each stage is shown in Figure 5.18. From the figure we see that for

Cross-rack skew	M/K=1.0	M/K =1.05	M/K =1.1
≤ 4	24.45	29.22	30.81
4 to 8	15.26	27.60	33.92
≥ 8	48.31	61.82	65.82

Table 5.4: Shuffle improvements with respect to baseline as cross-rack skew increases.

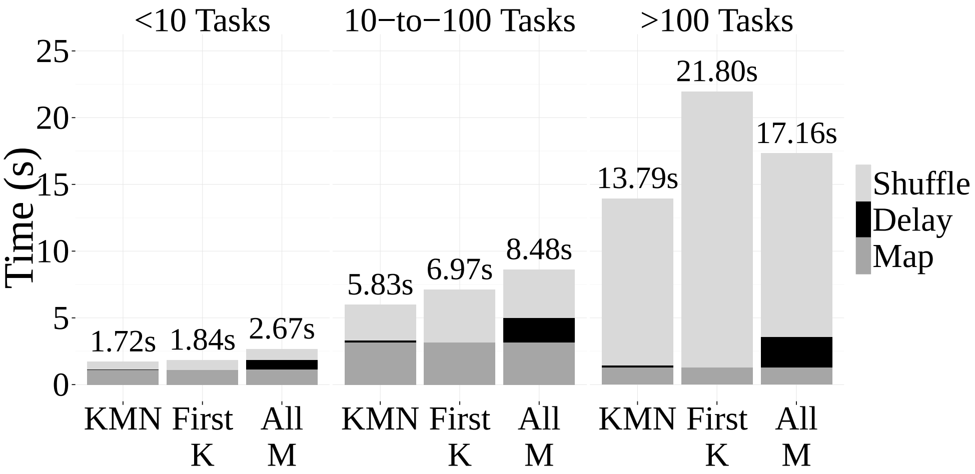


Figure 5.18: Benefits from straggler mitigation and delayed stage launch.

small jobs (< 10 tasks) which don't suffer from cross-rack skew, KMN performs similar to picking the first K map outputs. This is because in this case stragglers dominate the shuffle wins possible from using extra tasks. For larger tasks we see that our heuristic can dynamically adjust the stage delay to ensure we avoid stragglers while getting the benefits of balanced shuffle operations. For example for jobs with > 10 tasks KMN adds $5\% - 14\%$ delay after first K tasks complete and still gets most of the shuffle benefits. Overall, this results in an improvement of up to 35%.

For more fine-grained analysis we also ran an event-driven simulation that uses task completion times from the same Facebook trace. The CDF of extra map tasks used is shown in Figure 5.20, where we see that around 80% of the jobs wait for 5% or more map tasks. We also measured the time relative to when the first K map tasks finished and to normalize the delay across jobs we compute the relative wait time. Figure 5.19 shows the CDF of relative wait times and we see that the delay is less than 25% for 62% of the jobs. The simulation results again show that our relative delay is not very long and that job completion time can be improved when we use extra tasks available within a short delay.

Sensitivity to reducer placement

To evaluate the importance of reduce placement strategy, we compare the time taken for the shuffle stage for the round-robin strategy described in §5.4.2 against a greedy assignment strategy

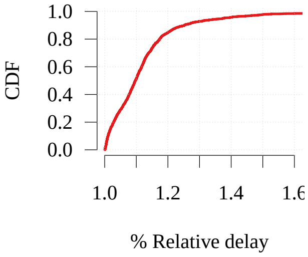


Figure 5.19: CDF of % time that the job was delayed

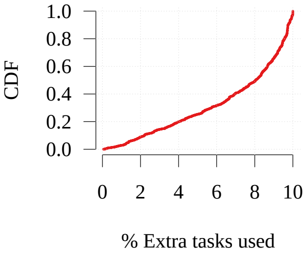


Figure 5.20: CDF of % of extra map tasks used.

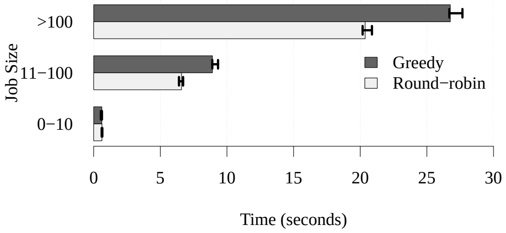


Figure 5.21: Difference between using greedy assignment of reducers versus using a round-robin scheme to place reducers among racks with upstream tasks.

that attempts to pack reducers into as few machines as possible. Note that the baseline used in our earlier experiments used a random reducer assignment policy and §5.5.2 compares the round-robin strategy to random assignment. Figure 5.21 shows the results from this experiment with the results broken down by job size. From the results we can see that for jobs with > 10 tasks using a round-robin placement can improve performance by 10%-30%. However for very small jobs, running tasks on more machines increases the variance and the greedy assignment in fact performs 8% better.

5.6 KMN Conclusion

The rapid growth of data stored in clusters, increasing demand for interactive analysis, and machine learning workloads have made it inevitable that applications will operate on subsets of data. It is therefore imperative that schedulers for cluster computing frameworks exploit the available choices to improve performance. As a step towards this goal we presented KMN, a system that improves data-aware scheduling for jobs with combinatorial choices. Using our prototype implementation, we see that KMN can improve performance by increasing locality and balancing intermediate data transfers.

Chapter 6

Future Directions & Conclusion

This thesis presented new system designs for large scale machine learning. We first studied the structure of machine learning workloads by considering a number of real world examples and extracting the significant properties that are important from a system design perspective. Following that we looked at how to understand performance characteristics of machine learning workloads by building performance models. In order to make it cheap to build performance models we developed Ernest, a tool that can be build performance models using just a few training data points. Finally we showed how using a simple performance model helps users make deployment decisions when using cloud computing infrastructure.

Analyzing the structure of machine learning workloads also helped us decompose the performance into two main parts: the control plane which performs coordination of tasks running across machines in a cluster and the data plane which dictates the time taken to transfer data and compute on it. Using this decomposition we looked at how we could add support to improve performance for machine learning workloads by making data processing schedulers aware of their structure.

In Drizzle we focused on low latency iterative workloads and, proposed group scheduling and pre-scheduling to minimize coordination overheads. We also studied how to balance the coordination overheads while ensuring that workloads remain adaptable to machine failures. We generalized our approach to stream processing workloads and showed how Drizzle is able to perform better than BSP-style systems like Spark and continuous processing systems like Flink.

To optimize the data plane we designed KMN, a data-aware scheduling framework that exploits the fact that machine learning workloads use sampling at each iteration. Based on this property we studied how we can improve locality while reading input data. We extended our scheme to allow for additional choice while performing shuffles and also discussed how we could avoid stragglers. Finally we also extended our design to accommodate approximate query processing workloads that sampled their input.

The techniques developed for this dissertation have been integrated into a number of software projects. Our scheme of improving reducer locality is now a part of the Apache Spark distribution and our straggler mitigation scheme from KMN is also used by machine learning libraries like Tensorflow [3]. The low latency scheduling techniques proposed in Drizzle have been integrated with BigDL, a deep learning framework [151] to reduce coordination overheads. Finally, our performance modeling approach of separating computation from communication was used to develop the cost-based optimizer in KeystoneML [157].

6.1 Future Directions

We next discuss some of the future directions on system design for machine learning especially in light of recent developments in algorithms and hardware.

Algorithm design. One of the main aspects of this thesis was that we tailored our system design to closely match the properties of the algorithms. Correspondingly there are new opportunities in designing algorithms while accounting for both the systems characteristics and the convergence properties. For example if we consider solving linear systems, the Gauss-Seidel method communicates more data per-iteration than the Jacobi method. However Gauss-Seidel has a better convergence rate than the Jacobi method and hence the appropriate algorithm to use depends on both the network bandwidth available and the condition number of the problem being solved.

A similar trade-off can also be seen in terms of sampling schemes. Recent work [163] shows that performing sampling with replacement improves convergence for block coordinate descent. However sampling with replacement is usually very expensive at large scale. Some early work [138] has shown how we can build upon Ernest, the performance prediction framework described in this thesis, and predict convergence rates of various algorithms.

Heterogeneous Hardware. The primary focus of this thesis was on compute clusters with homogeneous hardware. Our performance models were hence designed with minimal information about the hardware characteristics. With the growth in usage of GPUs [107], FPGAs [140] and other custom hardware devices [101] for deep learning algorithms, there are more complex design decisions in how one chooses the appropriate hardware for a given problem. We believe that our performance modeling techniques can be extended to include information about the hardware and thus help us jointly optimize hardware selection with algorithm design. Similarly with the widespread deployment of infiniband networks there are additional opportunities to encode network topology into our design.

Generalization vs. Specialization. A classic design choice in systems is between building a general purpose system that can be used across a number of applications as opposed to a specialized system that is tuned for a particular class of applications. General purpose systems typically make it easier to perform optimizations across workloads and simplify deployment. On the other hand specialized systems typically have better performance as they are tuned for the specific workload characteristics. In this thesis our approach was to pass through application characteristics to general purpose systems to achieve better performance. Recent systems [3] developed for algorithms like convolutional neural networks are specialized for their characteristics and lack support for features like elasticity found in existing general purpose frameworks. Following the design patterns we used in Drizzle and KMN, could help us further improve general purpose systems to handle new workload classes.

6.2 Concluding Remarks

With the growth of data collected across domains, machine learning methods play a key role in converting data into valuable information that can be used to power decisions. As the data sizes grow beyond what can be processed on a single machine, distributed data processing systems play a key role in building machine learning models. Our approach of designing better systems started from studying the execution properties of machine learning algorithms and characterizing how they are different from existing data processing workloads. Following that we generalized these properties to develop systems with better performance while maintaining resiliency and adaptability.

The design goal of software systems it to develop abstractions that can enable rapid development and good performance for commonly used workloads. Work towards this has led to the creation of many widely used file systems, database systems, operating systems etc. The advent of new workloads like large scale machine learning requires us to revisit the existing abstractions and design new systems that can both meet the requirements and exploit the flexibility offered by these workloads. This dissertation proposed system designs for training large scale machine learning models and we believe our approach of jointly considering algorithms and systems can be applied to other domains in the future.

Bibliography

[1] 1000 Genomes Project and AWS. http://aws.amazon.com/1000genomes/, 2014.

[2] Daniel J Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *VLDB*, 2003.

[3] Martin 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 *OSDI*, 2016.

[4] S. Acharya, P. Gibbons, and V. Poosala. Aqua: A fast decision support systems using approximate query answers. In *Proceedings of the International Conference on Very Large Data Bases*, pages 754–757, 1999.

[5] Vikram S Adve, John Mellor-Crummey, Mark Anderson, Ken Kennedy, Jhy-Chun Wang, and Daniel A Reed. An integrated compilation and performance analysis environment for data parallel programs. In *Supercomputing, 1995*, 1995.

[6] S. Agarwal, H. Milner, A. Kleiner, A. Talwarkar, M. Jordan, S. Madden, B. Mozafari, and I. Stoica. Knowing When YouâĂŹre Wrong: Building Fast and Reliable Approximate Query Processing Systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of data*. ACM, 2014.

[7] 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 *Proceedings of the 8th European conference on Computer Systems*. ACM, 2013.

[8] Sameer Agarwal, Anand P Iyer, Aurojit Panda, Samuel Madden, Barzan Mozafari, and Ion Stoica. Blink and it's done: interactive queries on very large data. *Proceedings of the VLDB Endowment*, 5(12):1902–1905, 2012.

[9] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM 2008*, Seattle, WA, 2008.

[10] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: coping with skewed content popularity in mapreduce clusters. In *Eurosys 2011*, pages 287–300, 2011.

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

[12] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Disk-locality in datacenter computing considered irrelevant. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, pages 12–12. USENIX Association, 2011.

[13] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: Attack of the clones. In *NSDI*, 2013.

[14] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. Pacman: Coordinated memory caching for parallel jobs. In *NSDI*, 2012.

[15] Ganesh Ananthanarayanan, Michael Chien-Chun Hung, Xiaoqi Ren, Ion Stoica, Adam Wierman, and Minlan Yu. Grass: Trimming stragglers in approximation analytics. In *NSDI*, 2014.

[16] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4):357–390, 1997.

[17] Apache Hadoop NextGen MapReduce (YARN). Retrieved 9/24/2013, URL: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html, 2013.

[18] Apache Mahout. http://mahout.apache.org/, 2014.

[19] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy H Katz, Andrew Konwinski, Gunho Lee, David A Patterson, Ariel Rabkin, Ion Stoica, et al. Above the clouds: A berkeley view of cloud computing. Technical report, Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, 2009.

[20] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark SQL: Relational data processing in Spark. In *SIGMOD*, 2015.

[21] Vasanth Balasundaram, Geoffrey Fox, Ken Kennedy, and Ulrich Kremer. A static performance estimator to guide data partitioning decisions. In *Symposium on Principles and Practice of Parallel Programming*, PPOPP ’91, pages 213–223, Williamsburg, Virginia, USA, 1991.

[22] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Supercomputing*, pages 340–347, 1997.

[23] Timo Bingmann, Michael Axtmann, Emanuel Jöbstl, Sebastian Lamm, Huyen Chau Nguyen, Alexander Noe, Sebastian Schlag, Matthias Stumpp, Tobias Sturm, and Peter Sanders. Thrill: High-performance algorithmic distributed batch data processing with c++. *CoRR*, abs/1608.05634, 2016.

[24] Blaise Barney. Message Passing Interface. https://computing.llnl.gov/tutorials/mpi/, 2016.

[25] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[26] Peter Bodík, Ishai Menache, Mosharaf Chowdhury, Pradeepkumar Mani, David A Maltz, and Ion Stoica. Surviving failures in bandwidth-constrained datacenters. In *Proceedings of ACM SIGCOMM 2012*, pages 431–442. ACM, 2012.

[27] Peter A Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, volume 5, pages 225–237, 2005.

[28] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT'2010)*, pages 177–187, Paris, France, August 2010. Springer.

[29] Léon Bottou and Olivier Bousquet. The tradeoffs of large scale learning. In *Advances in Neural Information Processing Systems*, pages 161–168, 2008.

[30] 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 *OSDI*, 2014.

[31] L. S. Brakmo and L. L. Peterson. TCP Vegas: End to End congestion avoidance on a global Internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, October 1995.

[32] Thorsten Brants, Ashok C. Popat, Peng Xu, Franz J. Och, and Jeffrey Dean. Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 858–867, Prague, Czech Republic, June 2007.

[33] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Supercomputing 2000*, Dallas, Texas, USA, 2000.

[34] Michael Cafarella, Edward Chang, Andrew Fikes, Alon Halevy, Wilson Hsieh, Alberto Lerner, Jayant Madhavan, and S. Muthukrishnan. Data management projects at Google. *SIGMOD Record*, 37(1):34–38, March 2008.

[35] Joseph P Campbell Jr and Douglas A Reynolds. Corpora for the evaluation of speaker recognition systems. In *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 2, pages 829–832, 1999.

[36] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. *CoRR*, abs/1506.08603, 2015.

[37] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *VLDB*, pages 1265–1276, 2008.

[38] Bradford L Chamberlain, Chong Lin, Sung-Eun Choi, Lawrence Snyder, E Christopher Lewis, and W Derrick Weathersby. Zpl's wysiwyg performance model. In *Third International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 50–61, 1998.

[39] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert Henry, Robert Bradshaw, and Nathan. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *PLDI*, 2010.

[40] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J Franklin, Joseph M Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R Madden, Fred Reiss, and Mehul A Shah. TelegraphCQ: continuous dataflow processing. In *SIGMOD*. ACM, 2003.

[41] K Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[42] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011.

[43] Surajit Chaudhuri, Gautam Das, and Utkarsh Srivastava. Effective use of block-level sampling in statistics estimation. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 287–298. ACM, 2004.

[44] Surajit Chaudhuri, Vivek Narasayya, and Ravishankar Ramamurthy. Estimating progress of execution for SQL queries. In *SIGMOD 2004*, pages 803–814, 2004.

[45] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. One billion word benchmark for measuring progress in statistical language modeling. Technical report, Google, 2013.

[46] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive analytical processing in big data systems: a cross-industry study of mapreduce workloads. *Proceedings of the VLDB Endowment*, 5(12):1802–1813, 2012.

[47] Mosharaf Chowdhury, Srikanth Kandula, and Ion Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *SIGCOMM 2013*, volume 43, pages 231–242, 2013.

[48] Mosharaf Chowdhury and Ion Stoica. Coflow: a networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 31–36, 2012.

[49] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. In *SIGCOMM*, 2011.

[50] Cisco. The Zettabyte Era: Trends and Analysis. Cisco Technology White Paper, May 2015. http://goo.gl/IMLY3u.

[51] R Clint Whaley, Antoine Petitet, and Jack J Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1):3–35, 2001.

[52] Adam Coates and Andrew Y Ng. Learning feature representations with k-means. In *Neural Networks: Tricks of the Trade*, pages 561–580. Springer, 2012.

[53] Gerard Cornuejols, George L Nemhauser, and Lairemce A Wolsey. The uncapacitated facility location problem. Technical report, DTIC Document, 1983.

[54] Tathagata Das, Yuan Zhong, Ion Stoica, and Scott Shenker. Adaptive stream processing using dynamic batch sizing. In *SOCC*, 2014.

[55] Extending the Yahoo! Streaming Benchmark. http://data-artisans.com/extending-the-yahoo-streaming-benchmark.

[56] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, A. Senior, P. Tucker, K. Yang, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems 25*, pages 1232–1240, 2012.

[57] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1), 2008.

[58] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[59] Ofer Dekel, Ran Gilad-Bachrach, Ohad Shamir, and Lin Xiao. Optimal distributed online prediction using mini-batches. *JMLR*, 13(1):165–202, January 2012.

[60] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *ASPLOS 2014*, pages 127–144, 2014.

[61] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM Journal on Scientific Computing*, 34(1):A206–A239, 2012.

[62] Bradley Efron. *The jackknife, the bootstrap and other resampling plans*, volume 38. SIAM, 1982.

[63] Facebook Presto. Retrieved 9/21/2013, URL: http://gigaom.com/2013/06/06/facebook-unveils-presto-engine-for-querying-250-pb-data-warehouse/, 2013.

[64] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. Liblinear: A library for large linear classification. *The Journal of Machine Learning Research*, 9:1871–1874, 2008.

[65] Tomás Feder and Daniel Greene. Optimal algorithms for approximate clustering. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 434–444, Chicago, Illinois, USA, 1988. ACM.

[66] Andrew D Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Eurosys 2012*, pages 99–112, 2012.

[67] R. Freitas, J. Slember, W. Sawdon, and L. Chiu. GPFS scans 10 billion files in 43 minutes. *IBM Advanced Storage Laboratory, San Jose, CA*, 2011.

[68] JF Gantz and D Reinsel. Digital universe study: Extracting value from chaos, 2011.

[69] GenBase repository. https://github.com/mitdbg/genbase, 2016.

[70] 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.

[71] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *Proceedings of the 8th European conference on Computer Systems*. ACM, 2013.

[72] J Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proc. of the 10th USENIX conference on Operating systems design and implementation*, 2012.

[73] Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. In *SIGMOD*, pages 102–111, 1990.

[74] Michael Grant and Stephen Boyd. Graph implementations for nonsmooth convex programs. In V. Blondel, S. Boyd, and H. Kimura, editors, *Recent Advances in Learning and Control*, Lecture Notes in Control and Information Sciences, pages 95–110. Springer-Verlag Limited, 2008. http://stanford.edu/~boyd/graph_dcp.html.

[75] Michael Grant and Stephen Boyd. CVX: Matlab software for disciplined convex programming, version 2.1. http://cvxr.com/cvx, March 2014.

[76] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP*, pages 202–210, 1989.

[77] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A Scalable and Flexible Data Center Network. In *ACM SIGCOMM 2009*, Barcelona, Spain, 2009.

[78] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues don't matter when you can jump them! In *NSDI*, 2015.

[79] Hadoop History Server REST APIs. http://archive.cloudera.com/cdh4/cdh/4/hadoop/hadoop-yarn/hadoop-yarn-site/HistoryServerRest.html.

[80] MapReduce Tutorial. hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html, 2014.

[81] Alon Halevy, Peter Norvig, and Fernando Pereira. The unreasonable effectiveness of data. *Intelligent Systems, IEEE*, 24(2):8–12, 2009.

[82] Nathan Halko, Per-Gunnar Martinsson, and Joel A Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, 53(2):217–288, 2011.

[83] Bingsheng He, Mao Yang, Zhenyu Guo, Rishan Chen, Wei Lin, Bing Su, Hongyi Wang, and Lidong Zhou. Wave computing in the cloud. In *HotOS*, 2009.

[84] Jun He, Duy Nguyen, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Reducing file system tail latencies with chopper. In *FAST*, pages 119–133, Santa Clara, CA, 2015.

[85] Herodotos Herodotou and Shivnath Babu. Profiling, what-if analysis, and cost-based optimization of MapReduce programs. *VLDB*, 4(11):1111–1122, 2011.

[86] Herodotos Herodotou, Fei Dong, and Shivnath Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *SOCC 2011*, 2011.

[87] 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*, volume 11, pages 261–272, 2011.

[88] 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 *NSDI*, 2011.

[89] Botong Huang, Matthias Boehm, Yuanyuan Tian, Berthold Reinwald, Shirish Tatikonda, and Frederick R. Reiss. Resource elasticity for large-scale machine learning. In *SIGMOD*, 2015.

[90] Po-Sen Huang, Haim Avron, Tara N Sainath, Vikas Sindhwani, and Bhuvana Ramabhadran. Kernel methods match deep neural networks on timit. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 1, page 6, 2014.

[91] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Eurosys*, 2007.

[92] Michael Isard and Martín Abadi. Falkirk wheel: Rollback recovery for dataflow systems. *arXiv preprint arXiv:1503.08877*, 2015.

[93] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *SOSP*, 2009.

[94] Van Jacobson. Congestion avoidance and control. *ACM SIGCOMM Computer Communication Review*, 18(4):314–329, 1988.

[95] Virajith Jalaparti, Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Bridging the tenant-provider gap in cloud services. In *SOCC*, 2012.

[96] Why is Thread.stop deprecated. http://docs.oracle.com/javase/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html, 1999.

[97] 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 *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[98] Wesley M Johnston, JR Hanna, and Richard J Millar. Advances in dataflow programming languages. *ACM Computing Surveys (CSUR)*, 36(1):1–34, 2004.

[99] E. Jonas, M. G. Bobra, V. Shankar, J. T. Hoeksema, and B. Recht. Flare Prediction Using Photospheric and Coronal Image Data. *ArXiv e-prints*, August 2017.

[100] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python. http://www.scipy.org/, 2001–.

[101] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *ISCA*, 2017.

[102] Rafal Józefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *CoRR*, abs/1602.02410, 2016.

[103] Qifa Ke, Michael Isard, and Yuan Yu. Optimus: a dynamic rewriting framework for data-parallel execution plans. In *Eurosys*, pages 15–28, 2013.

[104] P. Kolari, A. Java, T. Finin, T. Oates, and A. Joshi. Detecting spam blogs: A machine learning approach. In *Proceedings of the National Conference on Artificial Intelligence*, volume 21, page 1351, 2006.

[105] Tim Kraska, Ameet Talwalkar, John Duchi, Rean Griffith, Michael J Franklin, and Michael Jordan. MLbase: A distributed machine-learning system. In *Conference on Innovative Data Systems Research (CIDR)*, 2013.

[106] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *NetDB*, 2011.

[107] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012.

[108] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *SIGMOD*, pages 239–250, 2015.

[109] YongChul Kwon, Kai Ren, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Managing skew in hadoop. *IEEE Data Engineering Bulletin*, 36(1):24–33, 2013.

[110] John Langford. The Ideal Large-Scale Machine Learning Class. http://hunch.net/?p=1729.

[111] Nikolay Laptev, Kai Zeng, and Carlo Zaniolo. Early accurate results for advanced analytics on mapreduce. *Proceedings of the VLDB Endowment*, 5(10):1028–1039, 2012.

[112] David D Lewis, Yiming Yang, Tony G Rose, and Fan Li. Rcv1: A new benchmark collection for text categorization research. *The Journal of Machine Learning Research*, 5:361–397, 2004.

[113] 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 *OSDI*, pages 583–598, 2014.

[114] Mu Li, David G Andersen, Alex J Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. In *Advances in Neural Information Processing Systems*, pages 19–27, 2014.

[115] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 661–670. ACM, 2014.

[116] Wei Lin, Zhengping Qian, Junwei Xu, Sen Yang, Jingren Zhou, and Lidong Zhou. Streamscope: continuous reliable distributed processing of big data streams. In *NSDI*, pages 439–453, 2016.

[117] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 2012.

[118] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, et al. The Stanford CoreNLP natural language processing toolkit. In *ACL*, 2014.

[119] Omid Mashayekhi, Hang Qu, Chinmayee Shah, and Philip Levis. Scalable, fast cloud computing with execution templates. *CoRR*, abs/1606.01972, 2016.

[120] Julian McAuley, Rahul Pandey, and Jure Leskovec. Inferring networks of substitutable and complementary products. In *KDD*, pages 785–794, 2015.

[121] John McCalpin. STREAM update for Intel Xeon Phi SE10P. `http://www.cs.virginia.edu/stream/stream_mail/2013/0015.html`, 2013.

[122] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[123] Frank McSherry, Michael Isard, and Derek G Murray. Scalability! but at what cost? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[124] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB 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. *Journal of Machine Learning Research*, 17(34):1–7, 2016.

[125] Machine learning, microsoft azure. http://azure.microsoft.com/en-us/services/machine-learning/, 2016.

[126] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[127] Kristi Morton, Magdalena Balazinska, and Dan Grossman. Paratimer: A progress indicator for mapreduce dags. In *SIGMOD 2010*, pages 507–518, Indianapolis, Indiana, USA, 2010.

[128] 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.

[129] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *SOSP*, pages 439–455, 2013.

[130] Derek G Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. Ciel: a universal execution engine for distributed data-flow computing. In *NSDI*, pages 113–126, 2011.

[131] Dushyanth Narayanan. *Operating System Support for Mobile Interactive Applications*. PhD thesis, CMU, 2002.

[132] Oliver Niehorster, Alexander Krieger, Jens Simon, and Andre Brinkmann. Autonomic resource management with support vector machines. In *International Conference on Grid Computing (GRID '11)*, pages 157–164, 2011.

[133] Frank Austin Nothaft, Matt Massie, Timothy Danford, Zhao Zhang, Uri Laserson, Carl Yeksigian, Jey Kottalam, Arun Ahuja, Jeff Hammerbacher, Michael Linderman, Michael J. Franklin, Anthony D. Joseph, and David A. Patterson. Rethinking data-intensive science using scalable analytics systems. In *SIGMOD*, pages 631–646, 2015.

[134] L. Ortiz, VT de Almeida, and M. BalazinskaâĂă. Changing the Face of Database Cloud Services with Personalized Service Level Agreements. In *Proceedings of the International Conference on Innovative Data Systems Research (CIDR)*, 2015.

[135] 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. In *HotOS*, 2013.

[136] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *SOSP*, pages 69–84, 2013.

[137] Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. The quantcast file system. *Proceedings of the VLDB Endowment*, 2013.

[138] Xinghao Pan, Shivaram Venkataraman, Zizheng Tai, and Joseph Gonzalez. Hemingway: Modeling Distributed Optimization Algorithms. In *Machine Learning Systems Workshop (Co-located with NIPS)*, 2016.

[139] Friedrich Pukelsheim. *Optimal design of experiments*, volume 50. SIAM, 1993.

[140] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Eric Peterson, Aaron Smith, Jason Thong, Phillip Yi Xiao, Doug Burger, Jim Larus, Gopi Prashanth Gopal, and Simon Pope. A reconfigurable fabric for accelerating large-scale datacenter services. In *ISCA*, 2014.

[141] Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. In *Advances in neural information processing systems*, pages 1177–1184, 2007.

[142] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *HPDC 1998*, 1998.

[143] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. HOGWILD!: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[144] Sebastian Schelter, Stephan Ewen, Kostas Tzoumas, and Volker Markl. All roads lead to rome: optimistic recovery for distributed iterative data processing. In *CIKM*, 2013.

[145] Nicol Schraudolph, Jin Yu, and Simon Günter. A stochastic quasi-newton method for online convex optimization. *Journal of Machine Learning Research*, 2:428–435, 2007.

[146] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.

[147] Shai Shalev-Shwartz and Tong Zhang. Stochastic dual coordinate ascent methods for regularized loss. *The Journal of Machine Learning Research*, 14(1):567–599, 2013.

[148] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400, 2011.

[149] Juwei Shi, Jia Zou, Jiaheng Lu, Zhao Cao, Shiqiang Li, and Chen Wang. MRTuner: A Toolkit to Enable Holistic Optimization for MapReduce Jobs. *VLDB*, 7(13), 2014.

[150] Piyush Shivam, Varun Marupadi, Jeff Chase, Thileepan Subramaniam, and Shivnath Babu. Cutting corners: workbench automation for server benchmarking. In *USENIX ATC*, pages 241–254, 2008.

[151] Shivaram Venkataraman. Accelerating deep learning training with bigdl and drizzle on apache spark. https://rise.cs.berkeley.edu/blog/accelerating-deep-learning-training-with-bigdl-and-drizzle-on-apache-s

[152] L. Sidirourgos, ML Kersten, and P. Boncz. Sciborq: Scientific Data Management with Bounds on Runtime and Quality. In *Proceedings of the International Conference on Innovative Data Systems Research (CIDR)*, pages 296–301, 2011.

[153] SÃűren Sonnenburg and Vojtech Franc. Coffin: A computational framework for linear svms. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 999–1006, 2010.

[154] Apache Spark: Monitoring and Instrumentation. http://spark.apache.org/docs/latest/monitoring.html.

[155] Apache Spark: Submitting Applications. http://spark.apache.org/docs/latest/submitting-applications.html, 2014.

[156] Evan Sparks. Announcing KeystoneML. https://amplab.cs.berkeley.edu/announcing-keystoneml, 2015.

[157] Evan R Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J Franklin, and Benjamin Recht. Keystoneml: Optimizing pipelines for large-scale advanced analytics. In *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*, pages 535–546. IEEE, 2017.

[158] Lincoln D Stein et al. The case for cloud computing in genome informatics. *Genome Biology*, 11(5):207, 2010.

[159] P.V. Sukhatme and B.V. Sukhatme. *Sampling theory of surveys: with applications*. Asia Publishing House, 1970.

[160] Rebecca Taft, Manasi Vartak, Nadathur Rajagopalan Satish, Narayanan Sundaram, Samuel Madden, and Michael Stonebraker. Genbase: A complex analytics genomics benchmark. In *SIGMOD*, pages 177–188, 2014.

[161] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

[162] Stephen Tu, Rebecca Roelofs, Shivaram Venkataraman, and Benjamin Recht. Large scale kernel learning using block coordinate descent. *arXiv preprint arXiv:1602.05310*, 2016.

[163] Stephen Tu, Shivaram Venkataraman, Ashia C. Wilson, Alex Gittens, Michael I. Jordan, and Benjamin Recht. Breaking locality accelerates block Gauss-Seidel. In *ICML*, pages 3482–3491, 2017.

[164] Apache Spark, Preparing for the Next Wave of Reactive Big Data. http://goo.gl/FqEh94, 2015.

[165] Amin Vahdat, Mohammad Al-Fares, Nathan Farrington, Radhika Niranjan Mysore, George Porter, and Sivasankar Radhakrishnan. Scale-Out Networking in the Data Center. *IEEE Micro*, 30(4):29–41, July 2010.

[166] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[167] Robert A Van De Geijn and Jerrell Watts. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency-Practice and Experience*, 9(4):255–274, 1997.

[168] Shivaram Venkataraman, Aurojit Panda, Ganesh Ananthanarayanan, Michael J Franklin, and Ion Stoica. The power of choice in data-aware cluster scheduling. In *OSDI 2014*, pages 301–316, 2014.

[169] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 374–389. ACM, 2017.

[170] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: efficient performance prediction for large-scale advanced analytics. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, pages 363–378. USENIX Association, 2016.

[171] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. Aria: Automatic resource inference and allocation for mapreduce environments. In *ICAC 2011*, pages 235–244, Karlsruhe, Germany, 2011.

[172] Abhishek Verma, Brian Cho, Nicolas Zea, Indranil Gupta, and Roy H Campbell. Breaking the mapreduce stage barrier. *Cluster computing*, 16(1):191–206, 2013.

[173] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Eurosys*, 2015.

[174] Jens Vygen. Approximation algorithms facility location problems. Technical Report 05950, Research Institute for Discrete Mathematics, University of Bonn, 2005.

[175] Wenting Wang, Le Xu, and Indranil Gupta. Scale up Vs. Scale out in Cloud Storage and Graph Processing System. In *Proceedings of the 2nd IEEE Workshop on Cloud Analytics*, 2015.

[176] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Managed Communication and Consistency for Fast Data-parallel Iterative Analytics. In *SOCC*, pages 381–394, 2015.

[177] Reynold Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: SQL and Rich Analytics at Scale. In *SIGMOD 2013*, 2013.

[178] Neeraja J. Yadwadkar, Ganesh Ananthanarayanan, and Randy Katz. Wrangler: Predictable and faster jobs using fewer resources. In *SOCC*, 2014.

[179] Benchmarking Streaming Computation Engines at Yahoo! https://yahooeng.tumblr.com/post/135321837876.

[180] Wucherl Yoo, Kevin Larson, Lee Baugh, Sangkyum Kim, and Roy H. Campbell. Adp: Automated diagnosis of performance pathologies using hardware events. In *International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2012*, pages 283–294, London, England, UK, 2012.

[181] Hsiang-Fu Yu, Hung-Yi Lo, Hsun-Ping Hsieh, Jing-Kai Lou, Todd G McKenzie, Jung-Wei Chou, Po-Han Chung, Chia-Hua Ho, Chun-Fu Chang, Yin-Hsuan Wei, et al. Feature engineering and classifier ensemble for KDD Cup 2010. *KDD Cup 2010*, 2010.

[182] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P.K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 1–14, 2008.

[183] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Eurosys*, 2010.

[184] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[185] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10. USENIX Association, 2010.

[186] M. Zaharia, A. Konwinski, A.D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation*, pages 29–42, 2008.

[187] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, 2013.

[188] Zhao Zhang, Kyle Barbary, Frank Austin Nothaft, Evan R. Sparks, Oliver Zahn, Michael J. Franklin, David A. Patterson, and Saul Perlmutter. Scientific computing meets big data technology: An astronomy use case. *CoRR*, abs/1507.03325, 2015.

[189] Huasha Zhao and John Canny. Kylix: A sparse allreduce for commodity clusters. In *43rd International Conference on Parallel Processing (ICPP)*, pages 273–282, 2014.

[190] Fang Zheng, Guoliang Zhang, and Zhanjiang Song. Comparison of different implementations of MFCC. *Journal of Computer Science and Technology*, 16(6):582–589, 2001.