Optimizing Timeliness, Accuracy, and Cost in Geo-Distributed Data-Intensive Computing Systems

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Benjamin Heintz

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor of Philosophy

Professor Abhishek Chandra

December, 2016

© Benjamin Heintz 2016
ALL RIGHTS RESERVED

Acknowledgements

I would like to thank Chenyu Wang for developing and running the experiments in Chapter 3, and Ravali Kandur for her assistance deploying Apache Storm on PlanetLab.

The work presented here was supported in part by NSF Grants CNS-0643505, CNS-0519894, CNS-1413998, and IIS-0916425, by an IBM Faculty Award, and by a University of Minnesota Doctoral Dissertation Fellowship.

About six years ago, I set my sights on an M.S. degree in Computer Science. As I was beginning to plan for what I thought would be a brief career as a graduate student, I met with Prof. Jaideep Srivastava to discuss possible summer research. I am incredibly thankful that, instead of talking about which summer projects I could work on, he encouraged me to aim higher and pursue a Ph.D.

I owe a tremendous debt of gratitude to my collaborators, especially Prof. Jon Weissman, Prof. Ramesh K. Sitaraman, and my advisor, Prof. Abhishek Chandra. If I have developed even a small fraction of their ability to see the details without losing track of the big picture, to write well, and most of all to be persistent, then the last five years will have been worth it. I am particularly grateful for my advisor's endless patience and unshakable calm. Some of the most important results in my research have only come together after I have nearly given up on a deadline or called off a set of experiments. I am grateful that he never let me off the hook so easily.

On a personal note, I cannot thank my parents Henry and Jayne Heintz enough for their support, especially through college and graduate school. I would not have access to a sliver of the opportunities that I do if it weren't for them.

Finally, thanks to my wife Anne, for always being there for me, even when that means listening to me complain about how my experiments aren't working.

Dedication

To my dad.

Abstract

Big Data touches every aspect of our lives, from the way we spend our free time to the way we make scientific discoveries. Netflix streamed more than 42 billion hours of video in 2015, and in the process recorded massive volumes of data to inform video recommendations and plan investments in new content. The CERN Large Hadron Collider produces enough data to fill more than one billion DVDs *every week*, and this data has led to the discovery of the Higgs boson particle.

Such large scale computing is challenging because no one machine is capable of ingesting, storing, or processing all of the data. Instead, applications require distributed systems comprising many machines working in concert. Adding to the challenge, many data streams originate from geographically distributed sources. Scientific sensors such as LIGO span multiple sites and generate data too massive to process at any one location. The machines that analyze these data are also geo-distributed; for example Netflix and Facebook users span the globe, and so do the machines used to analyze their behavior.

Many applications need to process geo-distributed data on geo-distributed systems with low latency. A key challenge in achieving this requirement is determining *where* to carry out the computation. For applications that process unbounded data streams, two performance metrics are critical: *WAN traffic* and *staleness* (i.e., delay in receiving results). To optimize these metrics, a system must determine *when to communicate results* from distributed resources to a central data warehouse.

As an additional challenge, constrained WAN bandwidth often renders exact computation infeasible. Fortunately, many applications can tolerate inaccuracy, albeit with diverse preferences. To support diverse applications, systems must determine *what partial results to communicate* in order to achieve the desired staleness-error tradeoff.

This thesis presents answers to these three questions—where to compute, when to communicate, and what partial results to communicate—in two contexts: *batch* computing, where the complete input data set is available prior to computation; and *stream* computing, where input data are continuously generated. We also explore the challenges facing emerging programming models and execution engines that unify stream and batch computing.

Contents

Acknowledgements **i**

Dedication **ii**

Abstract **iii**

List of Tables **ix**

List of Figures **x**

1 Introduction **1**

1.1 Challenges in Geo-Distributed Data-Intensive Computing 2

1.1.1 Where to Place Computation 3

1.1.2 When to Communicate Partial Results 4

1.1.3 What Partial Results to Communicate 5

1.2 Summary of Research Contributions 5

1.2.1 Batch Processing 5

1.2.2 Stream Processing 7

1.2.3 Unified Stream & Batch Processing 8

1.3 Outline 9

2 Model-Driven Optimization of MapReduce Applications **10**

2.1 Introduction 10

2.1.1 The MapReduce Framework and an Optimization Example . . . 11

2.1.2 Research Contributions 13

2.2 Model and Optimization 14
2.2.1 Model 14
2.2.2 Model Validation 20
2.2.3 Optimization Algorithm 25
2.3 Model and Optimization Results 27
2.3.1 Modeled Environments 27
2.3.2 Heuristics vs. Model-Driven Optimization 28
2.3.3 End-to-End vs. Myopic 31
2.3.4 Single-Phase vs. Multi-Phase 33
2.3.5 Barriers vs. Pipelining 34
2.3.6 Compute Resource Distribution 37
2.3.7 Input Data Distribution 37
2.4 Comparison to Hadoop 39
2.4.1 Experimental Setup 40
2.4.2 Applications 40
2.4.3 Experimental Results 41
2.5 Related Work 46
2.6 Concluding Remarks 47

3 Cross-Phase Optimization in MapReduce **48**
3.1 Introduction 48
3.2 Map-Aware Push 49
3.2.1 Overlapping Push and Map to Hide Latency 50
3.2.2 Overlapping Push and Map to Improve Scheduling 51
3.2.3 Map-Aware Push Scheduling 51
3.2.4 Implementation in Hadoop 52
3.2.5 Experimental Results 53
3.3 Shuffle-Aware Map 55
3.3.1 Shuffle-Aware Map Scheduling 56
3.3.2 Implementation in Hadoop 58
3.3.3 Experimental Results 58
3.4 Putting it all Together 60

3.4.1 Amazon EC2 60
3.4.2 PlanetLab 62
3.5 Related Work 63
3.6 Concluding Remarks 65

4 Optimizing Timeliness and Cost in Geo-Distributed Streaming Analytics 67
4.1 Introduction 67
4.1.1 Challenges in Optimizing Windowed Grouped Aggregation . . . 69
4.1.2 Research Contributions 70
4.2 Problem Formulation 72
4.3 Dataset and Workload 74
4.4 Minimizing WAN Traffic and Staleness 76
4.5 Practical Online Algorithms 80
4.5.1 Simulation Methodology 81
4.5.2 Emulating the Eager Optimal Algorithm 82
4.5.3 Emulating the Lazy Optimal Algorithm 85
4.5.4 The Hybrid Algorithm 88
4.6 Implementation 90
4.6.1 Edge 91
4.6.2 Center 92
4.7 Experimental Evaluation 93
4.7.1 Aggregation Using a Single Edge 93
4.7.2 Scaling to Multiple Edges 94
4.7.3 Effect of Laziness Parameter 96
4.7.4 Impact of Network Capacity 97
4.8 Discussion 98
4.8.1 Compression 98
4.8.2 Variable-Size Aggregates 100
4.8.3 Applicability to Other Environments 101
4.9 Related Work 102
4.10 Concluding Remarks 103

5 Trading Timeliness and Accuracy in Geo-Distributed Streaming Analytics 104
5.1 Introduction 104
5.2 Problem Formulation 106
5.2.1 Problem Statement 106
5.2.2 The Staleness-Error Tradeoff 107
5.3 Offline Algorithms 109
5.3.1 Minimizing Staleness (Error-Bound) 109
5.3.2 Minimizing Error (Staleness-Bound) 110
5.3.3 High-Level Lessons 115
5.4 Online Algorithms 116
5.4.1 The Two-Part Cache Abstraction 116
5.4.2 Error-Bound Algorithms 118
5.4.3 Staleness-Bound Algorithms 120
5.4.4 Implementation 123
5.5 Evaluation 123
5.5.1 Setup 124
5.5.2 Comparison Algorithms 124
5.5.3 Simulation Results: Minimizing Staleness (Error-Bound) 126
5.5.4 Simulation Results: Minimizing Error (Staleness-Bound) 128
5.5.5 Experimental Results 129
5.6 Discussion 138
5.6.1 Alternative Approximation Approaches 138
5.6.2 Coordination Between Multiple Edges 139
5.6.3 Incorporating WAN Latency 140
5.7 Related Work 140
5.8 Concluding Remarks 141

6 Challenges and Opportunities in Unified Stream & Batch Analytics 142
6.1 Introduction 142
6.2 Background 144
6.2.1 Diverged Stream and Batch Processing 144

6.2.2 Early Attempts at Unified Stream & Batch Processing 145
6.2.3 The State of the Art: Apache Beam 146
6.2.4 Challenges Magnified by Geo-Distribution 148
6.3 Simulation Setup 149
6.4 Programming Models 149
6.4.1 Convenience 150
6.4.2 Expressiveness 153
6.4.3 Research Challenges 155
6.5 Execution Engines 156
6.5.1 Hopping Windows 156
6.5.2 Out-of-Order Arrivals 169
6.6 Concluding Remarks 177

7 Conclusion 179

7.1 Research Contributions 180
7.1.1 Batch Processing 180
7.1.2 Stream Processing 180
7.1.3 Unified Stream & Batch Processing 181
7.2 Future Research Directions 182
7.2.1 Alternative Performance Metrics and Optimization Objectives . 182
7.2.2 Alternative Applications 183
7.2.3 Scalable System Implementations 184

References 185

List of Tables

2.1 Measured bandwidth (KBps) of the slowest/fastest links between nodes in each continent. 24

3.1 Measured bandwidths in the EC2 experimental setup. 54

3.2 Measured bandwidths in the PlanetLab experimental setup. 55

3.3 Number of map tasks assigned to each mapper node in our PlanetLab test environment. 62

4.1 Queries used throughout this thesis. 75

List of Figures

1.1 Distribution of log data volume from Akamai's download network for December 2010. The y-axis shows the fraction of log data generated in a given country (on log-scale); the x-axis shows the rank of that country in sorted order by volume. 3

2.1 An example of a two-cluster distributed environment, with one data source, one mapper, and one reducer in each cluster. The intra-cluster (*local*) communication links are solid lines, while the inter-cluster (*non-local*) communication links are the dotted lines. 12

2.2 A tripartite graph model for distributed MapReduce with 3 data sources, 2 mappers, and 2 reducers. 15

2.3 Measured vs. model-predicted makespan from the Hadoop implementation on the local emulated testbed. 26

2.4 A comparison of three optimization heuristics and our end-to-end multi-phase optimization (e2e multi). 30

2.5 Our end-to-end multi-phase optimization (e2e multi) compared to a uniform heuristic and a myopic optimization approach. 32

2.6 Our end-to-end multi-phase optimization (e2e multi) compared to a uniform heuristic as well as end-to-end push (e2e push) and end-to-end shuffle (e2e shuffle) optimization. 35

2.7 A comparison of barrier configurations. Makespan is normalized relative to the optimal makespan for an all-global-barrier configuration. Each bar represents the boundary at which the global barrier is relaxed to pipelining; "all" corresponds to an all-pipelining configuration. 36

2.8 Our end-to-end multi-phase optimization (e2e) compared to a uniform heuristic and myopic optimization for different network environments. Makespan is normalized relative to the uniform baseline. 38
2.9 Our end-to-end multi-phase optimization (e2e) compared to uniform and myopic baselines in the eight-node PlanetLab environment with inputs distributed according to Akamai logs. Makespan is normalized relative to the uniform baseline. 39
2.10 Actual makespan for three sample applications on our local testbed. Error bars reflect 95% confidence intervals. 42
2.11 Effect of Hadoop's dynamic scheduling mechanisms applied atop our optimized static execution plan for three sample applications. Error bars reflect 95% confidence intervals. 43
2.12 Effect of Hadoop's dynamic scheduling mechanisms applied atop a competitive Hadoop baseline plan. Error bars reflect 95% confidence intervals. 43
2.13 Effect of HDFS file replication for three sample applications. Error bars reflect 95% confidence intervals. 45
3.1 A simple example network with two data sources and two mappers. . . 50
3.2 Runtime of a Hadoop WordCount job on text data for the push-then-map approach and the *Map-Aware Push* approach on globally distributed Amazon EC2 and PlanetLab test environments. Error bars reflect 95% confidence intervals. 54
3.3 An example network where links from mapper C to reducers D and E are shuffle bottlenecks. 56
3.4 Runtime of the InvertedIndex job on eBook data for the default Hadoop scheduler and our *Shuffle-Aware Map* scheduler. Both approaches use an overlapped push and map in these experiments. Error bars reflect 95% confidence intervals. 59
3.5 Execution time for traditional Hadoop compared with our proposed *Map-Aware Push* and *Shuffle-Aware Map* techniques (together, *End-to-end*) for InvertedIndex and Sessionization applications on our EC2 test environment. Error bars reflect 95% confidence intervals. 61

3.6 Execution time for traditional Hadoop compared with our proposed *Map-Aware Push* and *Shuffle-Aware Map* techniques (together, *End-to-end*) for for the InvertedIndex application with 800 MB eBook data on our PlanetLab test environment. Error bars reflect 95% confidence intervals. 62
4.1 The distributed model for a typical analytics service comprises a single center and multiple edges, connected by a wide-area network. 68
4.2 Simple aggregation algorithms such as streaming and batching optimize one metric over the other and their performance depends on the data and query characteristics. 71
4.3 Staleness s is defined as the delay between the end of the window and final results for the window becoming available at the center. 73
4.4 Akamai Web download service data set characteristics. 77
4.5 Eager online algorithms. 84
4.6 Lazy online algorithms. 87
4.7 Sensitivity of the hybrid algorithms with a range of α values to overpredicting the available network capacity. Staleness is normalized by window length. 88
4.8 Cache size over time for eager and lazy offline optimal algorithms. Sizes are normalized relative to the largest size. 89
4.9 Average traffic for hybrid algorithms with several values of the laziness parameter α. Traffic is normalized relative to an optimal algorithm. . . 90
4.10 Aggregation is distributed over Apache Storm clusters at each edge as well as at the center. 91
4.11 Performance for batching, streaming, optimal, and our hybrid algorithm for the `large` query with a low stream arrival rate using a one-edge Apache Storm deployment on PlanetLab. 94
4.12 Performance for batching, streaming, optimal, and our hybrid algorithm for a range of queries and stream arrival rates using a three-edge Apache Storm deployment on PlanetLab. 95
4.13 Performance for batching, streaming, optimal, and our hybrid algorithm for the `large` query with low and high stream arrival rates using a six-edge Apache Storm deployment on PlanetLab. 96

4.14 Effect of laziness parameter α using a three-edge Apache Storm deployment on PlanetLab with query `large`. 97
4.15 Traffic and staleness for different algorithms over a range of network capacities. 98
4.16 Compression ratio and latency for Google Snappy compression applied at several granularities to our anonymized Akamai trace. 99
4.17 Size of exact and approximate set representations with and without compression. 100
5.1 The staleness-error tradeoff curve for a `Sum` aggregation over a single time window. Note that we normalize staleness relative to the window length and error relative to the maximum possible error for the window. 108
5.2 We view the network as a sequence of contiguous slots. Shaded slots are unavailable to the current window due to the previous window's staleness. 111
5.3 Smallest-potential-error-first ordering minimizes error. 112
5.4 The cache is dynamically partitioned into primary and secondary parts. Updates are flushed only from the primary cache. 117
5.5 Impact of primary cache eviction policy on staleness given an error bound. Note the logarithmic y-axis scale. 119
5.6 Impact of the initial prefix error prediction algorithm on staleness given error bound. Note the logarithmic axis scales. 121
5.7 The impact of primary cache sizing policy on error given staleness bound. Note the logarithmic axis scales. 122
5.8 Normalized staleness for various aggregations and error bounds. Note the logarithmic y-axis scales. 127
5.9 Normalized staleness for `Sum` aggregation with various error bounds under medium and high bandwidth. Note the logarithmic y-axis scales. 128
5.10 Normalized error for various aggregations and staleness bounds. Note the logarithmic axis scales. 130
5.11 Normalized error for `Sum` aggregation with various staleness bounds under medium and high bandwidth. Note the logarithmic y-axis scale. 131
5.12 Comparison of Random and Caching algorithms for various aggregation functions on a PlanetLab testbed. Note the logarithmic y-axis scale. . . 132

5.13 Comparison of Random and Caching algorithms for Sum aggregation with various error and staleness constraints on a local testbed. Note the logarithmic y-axis scale. 132
5.14 Comparison of Random and Caching algorithms for Max aggregation with various error and staleness constraints on a local testbed. Note the logarithmic y-axis scale. 133
5.15 Comparison of Random and Caching algorithms for Sum aggregation under dynamic workload and WAN bandwidth on a local testbed. Arrival rate increases during window 6 (dashed vertical line), and available bandwidth changes during window 20 (dotted vertical line). 134
5.16 Comparison of Random and Caching algorithms for Sum aggregation with various emulated WAN bandwidths on a local testbed. Note the logarithmic y-axis scale. 135
5.17 Comparison of Random and Caching algorithms for Max aggregation with various emulated WAN bandwidths on a local testbed. Note the logarithmic y-axis scale. 136
5.18 Comparison of Random, Random (2-part), and Caching algorithms for Sum and Max aggregations at medium WAN bandwidth. Note the logarithmic y-axis scale. 137
6.1 Staleness over time for a Sum aggregation under an error constraint. . . 152
6.2 Staleness over time for a Sum aggregation under a staleness constraint. . 154
6.3 Error over time for a Sum aggregation under a staleness constraint. . . . 154
6.4 Hopping windows have length W and advance at interval H. 156
6.5 Traffic and staleness for an exact Sum aggregation. 158
6.6 The parallel approach with (offline) duplicate elimination is suboptimal. 162
6.7 Traffic, staleness, and error for a Sum aggregation under an error constraint. 164
6.8 Traffic, staleness, and error for a Max aggregation under an error constraint. 166
6.9 Traffic, staleness, and error for a Sum aggregation under a staleness constraint of $\frac{W}{16}$. 168
6.10 Traffic, staleness, and error for a Sum aggregation under an error constraint, using a 99th-percentile heuristic watermark, for arrivals with mean delay of 1% of the window length. 172

6.11 Traffic, staleness, and error for a `Sum` aggregation under an error constraint for tumbling windows, using a 99$^{\text{th}}$-percentile heuristic watermark. 174
6.12 Traffic, staleness, and error for a `Sum` aggregation under an error constraint for tumbling windows, for arrivals with mean delay of 1% of the window length. 176

Chapter 1

Introduction

Big Data touches virtually every aspect of our lives, from the way we spend our free time, to the way we connect and share with friends and family, and even the way we make breakthrough scientific discoveries. For example, Netflix streamed more than 42 billion hours of video in 2015 [1], and in doing so, recorded trillions of events describing system and user behavior, which it used to recommend new videos, evaluate new features, and plan investment in new content [2].

Every day, more than a billion people around the world log onto Facebook and upload more than 350 million photos [3, 4]. These photos not only serve to connect users with one another, but also help teach massive computing systems how to understand images and make their descriptions accessible to visually impaired users [5]. Meanwhile, the CERN Large Hadron Collider produces enough data each week to fill more than one billion DVDs [6], and this data has led to the discovery of the Higgs boson particle, while massive data streams from LIGO [7] have enabled the detection of gravitational waves.

Computing at such massive scale is enormously challenging due to both the size and growth rate of data: No individual computer is capable of ingesting and storing all of the data, let alone performing the required computation. As a result, these applications require large distributed systems comprising many machines working in concert. Adding to the challenge is the fact that many interesting data streams originate from geographically distributed sources. For example, smart home sensor data and content delivery network (CDN) log data originate from sources that are physically fixed

at diverse locations all around the globe; such data are truly “born distributed” [8]. In scientific settings, sensors such as LIGO span several physical sites, and the data they generate are too massive to process at a single location. In many cases, the computing resources available to analyze these data are also geo-distributed. For example, Netflix and Facebook users span the globe, and so do the computers used to analyze and understand their behavior.

1.1 Challenges in Geo-Distributed Data-Intensive Computing

Many modern applications need to process geo-distributed data on geo-distributed resources. As a motivating example, consider a large content delivery network (CDN) such as Akamai. Content providers use CDNs to deliver web content, live and on-demand video, software downloads, and web applications to users around the world. The servers of a CDN are deployed in clusters in hundreds or even thousands of data centers around the world. Each server of a CDN records detailed data about each user that it interacts with: every web object that is served, each stream that is played, each application that is accessed, as well as each user action such as playing or pausing a stream. Besides user access information, each server also records network-level and system-level data such as network connection statistics [9]. In aggregate, the servers produce tens of billions of lines of log data originating from over a thousand locations each day. Figure 1.1 shows the truly distributed nature of log data from Akamai’s download network for the month of December 2010 where one can see that no individual country generated more than 14% of the log volume.

Analytics applications must process this massive geo-distributed data to extract detailed information, for example describing who is accessing the content, and from which networks and which geographies. Such applications present major challenges for researchers because communication over wide-area network links is often both slow and financially costly, and because both communication and computation resources are typically highly heterogeneous. The choice of *where* to place computation as well as *when* and precisely *what* to communicate therefore have pronounced effects on the timeliness, accuracy, and cost of geo-distributed computation.

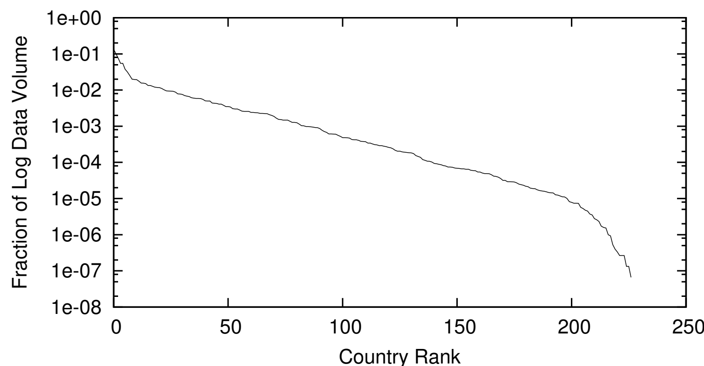


Figure 1.1: Distribution of log data volume from Akamai's download network for December 2010. The y-axis shows the fraction of log data generated in a given country (on log-scale); the x-axis shows the rank of that country in sorted order by volume.

1.1.1 Where to Place Computation

Geo-distributed data processing must be done in a manner that *minimizes the completion time (i.e., makespan)* so that analysts can quickly understand and act upon the key knowledge derived from their data. One major challenge in achieving low makespan when processing distributed data is determining *where* to carry out the computation. Jim Gray noted that "you can either move your questions or the data" [10]. These two options, however, represent two extreme possibilities. At one extreme, sending massive amounts of data from the diverse locations where the data originate to a single centralized location may be too slow [11] to meet the requirements of low latency. Further, the cost may be prohibitive, both in terms of data transfer costs as well as the need to build a single large dedicated data center for application processing. Finally, a centralized solution is intrinsically less fault-tolerant than a distributed one, since failure of the single centralized data center can cause a complete outage that is unacceptable for critical applications such as analytics and monitoring.

At the other extreme, if we map computation onto each input datum *in situ*, the results of these local subcomputations comprise intermediate data that must be aggregated or reorganized to generate the final analysis results. Unless such intermediate data are small relative to the input data, combining the intermediate data could be more costly than moving input data to a single centralized location. Further, the various locations may be highly *heterogeneous* in terms of their capacity to perform computation

and communicate results, leading to imbalanced resource utilization.

Between the two extremes of moving all the data to a centralized location for processing and moving all the computation to the sources of data, there is a rich continuum of possibilities that may in fact be more efficient than either extreme. We explore this continuum to optimize the completion time of data-intensive applications that process geo-distributed data.

1.1.2 When to Communicate Partial Results

When processing unbounded *streams* of data, it is common to logically subdivide inputs into *time windows* and produce results summarizing each of these windows. For instance, a web analyst may wish to count the total visits to her web site broken down by country and aggregated on an hourly basis to track content popularity. Similarly, a network operator may want to compute the average load in different parts of the network every five minutes to identify hotspots. In each case, the user would define a standing query that generates results periodically for each time window (every hour, five minutes, etc.).

Two key metrics are critical when performing such streaming computation in a geo-distributed setting: wide-area-network (WAN) *traffic* and *staleness* (the delay in receiving the result for a time window). In a typical geo-distributed analytics service, computation can be performed both at *edge* servers near the data sources and at a central data warehouse location. In this setting, the choice of *when to communicate* results from edges to the center is critical, as it impacts both traffic and staleness. For example, if inputs are only briefly aggregated at the edges before results are sent to the center, then staleness may be quite low, but excessive traffic will flow from the edges to the center. On the other hand, if inputs are aggregated at the edge for the full duration of the window, then traffic will be minimized, but staleness will be high due to the need to communicate a large batch of results at the end of the window.

As we will show, simple techniques fail to optimize both traffic and staleness at the same time, and instead pit these critical metrics against each other. Further, the factors influencing the performance of such algorithms are often hard to predict and can vary significantly over time, requiring the design of algorithms that can adapt to changing factors in a dynamic fashion.

1.1.3 What Partial Results to Communicate

As an additional challenge, constrained WAN bandwidth often renders exact computation with bounded staleness infeasible [12]. Fortunately, many real-world applications can tolerate some staleness or inaccuracy in their final results, albeit with diverse preferences. For instance, a network administrator may need to be alerted to potential network overloads *quickly* (within a few seconds or minutes), even if there is some degree of error in the results describing network load. On the other hand, a Web analyst might have only a small tolerance for error (say, <1%) in the application statistics (e.g., number of page hits), but be willing to wait for some time to obtain these results with the desired accuracy.

In order for a geo-distributed streaming analytics system to support this diverse range of application requirements, it must determine *what partial results to communicate* in order to achieve the desired staleness-error tradeoff. For example, if an application requires results with very low staleness, but can tolerate greater error, then it may be necessary to communicate early results that reflect only a partial view of the inputs. On the other hand, if an application demands a high degree of accuracy but can tolerate delay, then the results should be produced later, when a more complete view of inputs is available.

1.2 Summary of Research Contributions

This thesis presents systematic answers to three critical questions—where to compute, when to communicate, and what partial results to communicate—in two contexts: *batch* computing, where the full input data set is available prior to computation; and *stream* computing, where input data are continuously generated. We also explore the challenges and opportunities facing emerging programming models and execution engines that aim to unify stream and batch computing.

1.2.1 Batch Processing

In the batch processing context, we focus on the question of *where* to perform computation. We seek to minimize the execution time of applications in MapReduce [13], the de facto standard model for very-large-scale batch computing.

The main challenges here arise due to *heterogeneity*, as network links between machines exhibit diverse performance characteristics, and the machines themselves vary in their capacity to perform computation. The choice of where to compute is therefore critical: some network links and machines are much slower than others, and an efficient placement must avoid bottlenecks due to these slow resources while at the same time making effective use of all available resources.

Our work in this context follows a two-pronged approach: We first develop model-driven optimization techniques that serve as oracles and provide high-level insights, and then we apply these insights to develop practical techniques that we implement and demonstrate in a real-world MapReduce implementation.

Model-Driven Optimization

We devise a model to predict the execution time of MapReduce applications in geographically distributed environments as well as an optimization framework for determining the best placement of data and computation. We modify the popular Hadoop MapReduce framework to enact the placement decisions produced by our model-driven optimization, and we use this implementation to experimentally validate our model. We use our model and optimization to derive several insights, for example that application characteristics can significantly influence the optimal data and computation placement, and that our optimization yields greater benefits as the environment becomes increasingly distributed. Further, we show that our optimization approach can significantly outperform standard Hadoop based on experiments in a geographically distributed PlanetLab [14] testbed.

Practical Systems Techniques

Based on the insights from our model-driven optimization, we develop practical techniques for real-world MapReduce implementations such as Hadoop. In particular, we develop new cross-phase optimization techniques that optimize the data ingest (i.e., push) and map phases to enable push-map overlap and to allow map behavior to feed back into push dynamics. Similarly, we devise techniques that optimize the map and reduce phases to enable shuffle cost to feed back and affect map scheduling decisions.

We demonstrate the effectiveness of these techniques in reducing execution time for diverse applications through experiments on geographically distributed testbeds on both Amazon EC2 and PlanetLab.

1.2.2 Stream Processing

While batch computing is a fundamental part of many applications, stream computing continues to grow in importance. In many cases, analysts need to extract information from data in near-real-time. For example, systems operators at Netflix need to know within seconds where performance anomalies are occurring in order to avoid prolonged interruptions for their customers. Meanwhile, security analysts have little tolerance for delay in identifying possible intrusions. To support applications like these, systems consume continuous streams of data and produce summaries such as maximum or average values or trends over time.

Exact Computation

Our first contribution in the streaming context focuses on applications where exact results are required. A typical geo-distributed analytics service uses a simple hub-and-spoke model, comprising a single central data warehouse and multiple edges connected by a wide-area network (WAN). We examine the question of when to communicate results from the edges to the center in the context of *windowed grouped aggregation*, an important and widely used primitive in streaming analytics applications. We focus on designing algorithms to optimize two key metrics of any geo-distributed streaming analytics service: *WAN traffic* and *staleness* (the delay in getting the result). Towards this end, we present a family of optimal offline algorithms that *jointly minimize both staleness and traffic*. Using this as a foundation, we develop practical online aggregation algorithms based on the observation that grouped aggregation can be modeled as a *caching* problem where the cache size varies over time. This key insight allows us to exploit well known caching techniques in our design of online aggregation algorithms.

We demonstrate the practicality of these algorithms through an implementation in Apache Storm, deployed on a PlanetLab testbed. The results of our experiments, driven by workloads derived from anonymized traces of a popular web analytics service offered by a large commercial CDN, show that our online aggregation algorithms perform close

to the optimal offline algorithms for a variety of system configurations, stream arrival rates, and query types.

Approximate Computation

Due to limited WAN bandwidth, it is not always possible to produce exact results in near-real-time. When this is the case, applications must either sacrifice timeliness by allowing delayed—i.e., *stale*—results, or sacrifice accuracy by allowing some *error* in final results.

Continuing our focus on windowed grouped aggregation, we present optimal offline algorithms for minimizing staleness under an error constraint and for minimizing error under a staleness constraint. Using these offline algorithms as references, we present practical online algorithms for effectively trading off timeliness and accuracy under bandwidth limitations. Using a real-world workload, we demonstrate the effectiveness of our techniques through both trace-driven simulation as well as experiments on an Apache Storm-based implementation deployed on PlanetLab. Experiments demonstrate that our algorithms significantly reduce staleness and error relative to a practical random sampling/batching-based baseline across a diverse set of aggregation functions.

1.2.3 Unified Stream & Batch Processing

While both batch processing and stream processing are important topics on their own, applications are increasingly being developed that do not fit neatly into either of these extremes. For example, a network operator might rely on up-to-the-second summary data to identify potential performance anomalies, but then wish to compare this data against historical statistics. This application requires synthesizing results computed from both unbounded streams and bounded batches of data. As another example, an analyst who regularly computes a daily summary of an important dataset may discover that this report provides a useful signal for day-to-day decisions, and wish to accelerate the pace of reporting, generating hourly—or even continuous—reports. In other words, an application might evolve over time from processing bounded batches of data to processing an unbounded stream of data in near-real-time.

Historically, researchers have considered batch processing and stream processing to be separate problems, and distinct systems have been developed and optimized for

each of these problems. The current trend, however, is toward a unified programming model capable of expressing both batch and stream programs [15, 16], as well as unified computing engines capable of efficiently computing over both bounded batches and unbounded streams [17, 18].

The current state-of-the-art programming model requires application developers to explicitly answer the question of when to trigger the production of results. The best answer to this question, however, depends on the application-level requirements in terms of timeliness, accuracy, and cost, as well as dynamic factors such as network conditions and data characteristics. We argue that it is better to allow application developers to specify these high-level requirements and rely on systems to automatically determine when to produce results.

Although programming models raise challenging research questions on their own, they also highlight opportunities to extend our optimization techniques to better optimize timeliness, accuracy, and cost in emerging execution engines. We explore how the techniques presented in this thesis apply under these increasingly general programming models, and which challenges warrant further research attention.

1.3 Outline

The remainder of this thesis proceeds as follows. Chapter 2 presents our model-driven optimization techniques for determining where to place computation in order to minimize end-to-end execution time in batch processing settings. Chapter 3 presents practical systems techniques inspired by this model-driven optimization. In Chapter 4 we present our caching-based algorithms for determining when to communicate partial results in geo-distributed stream computing, and Chapter 5 generalizes these techniques to approximate computation. Chapter 6 explores the challenges and opportunities facing emerging programming models and execution engines for unified stream and batch computing. Finally, Chapter 7 concludes.

The work presented in this thesis also appears in workshop, conference, and journal publications [19, 20, 21, 22, 23].

Chapter 2

Model-Driven Optimization of MapReduce Applications

2.1 Introduction

A key question for efficient processing of geo-distributed data is *where* to carry out the computation. At one extreme, sending massive amounts of data from the diverse locations where the data originate to a single centralized location may be too slow [11] to meet the requirements of low latency. Further, the cost may be prohibitive, both in terms of data transfer costs as well as the need to build a single large dedicated data center for application processing. Finally, a centralized solution is intrinsically less fault-tolerant than a distributed one, since failure of the single centralized data center can cause a complete outage that is unacceptable for critical applications such as analytics and monitoring.

At the other extreme, if we map computation onto each input datum *in situ*, the results of these local subcomputations comprise intermediate data that must be aggregated or reorganized to generate the final analysis results. If such intermediate data are not small relative to the input data, then combining the intermediate data could be more costly than moving input data to a single centralized location. Further, the various locations may differ in their compute capacities, resulting in imbalanced resource utilization.

Between the two extremes of moving all the data to a centralized location for processing and moving all the computation to the sources of data, there is a rich continuum of possibilities that may in fact be more efficient than either extreme. This chapter focuses on exploring this continuum to optimize the completion time of data-intensive applications that process geographically distributed data. For the scope of this chapter, we limit our focus to a specific data analysis framework: *MapReduce* [13]. MapReduce is a powerful programming model used to program a wide variety of analytics applications, and has achieved popularity through its open-source implementation, Hadoop [24], as well as several analysis tools and programming languages built on top of Hadoop. A broad class of analytics applications of interest fit into the MapReduce application framework. Thus, we focus our attention on the efficient execution of applications that fit the MapReduce framework and explore the possibilities for speeding up such applications using a model-driven approach.

2.1.1 The MapReduce Framework and an Optimization Example

The MapReduce application framework comprises a simple but powerful programming model that processes data represented as *key-value pairs*. It incorporates a *map* function and a *reduce* function. The execution of the application proceeds in four phases: *push*, *map*, *shuffle*, and *reduce*. In the push phase, the input key-value pairs are moved from the data sources to the mappers. In the map phase, the map function is applied to each key-value pair, resulting in zero or more intermediate key-value pairs. In the shuffle phase, these intermediate pairs are grouped by key. Finally, in the reduce phase, the reduce function is applied to each intermediate key along with all of the values associated with that key, producing zero or more output key-value pairs.

Given an application that uses the MapReduce programming model, our work explores the space of options for executing the application on a distributed platform so as to minimize the completion time (i.e., *makespan*) of the application. Such an optimization involves exploring which servers of the distributed platform should process which portion of the data for both the map and the reduce operations.

We illustrate the importance of optimizing the data communication and task placement in a distributed environment with a simple example and show that the best approach depends on the platform and application characteristics.

Consider the example MapReduce platform shown in Figure 2.1. Assume that the data sources D_1 and D_2 have 150 GB and 50 GB of data, respectively. Let us model a parameter α that represents the ratio of the amount of data output by a mapper to its input; i.e., α is the expansion (or reduction) of the data in the map phase and is specific to the application. We model different situations below.

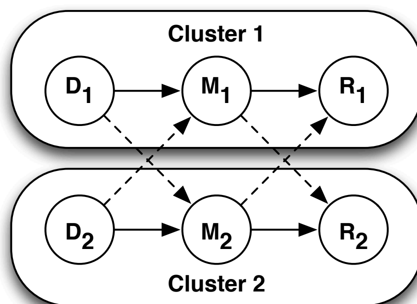


Figure 2.1: An example of a two-cluster distributed environment, with one data source, one mapper, and one reducer in each cluster. The intra-cluster (*local*) communication links are solid lines, while the inter-cluster (*non-local*) communication links are the dotted lines.

First, consider a situation where $\alpha = 1$ and the network is perfectly homogeneous. That is, all links have the same bandwidth—say 100 MBps each—whether they are local or non-local, and the computational resources at each mapper and reducer are homogeneous too; say each can process data at the rate of 100 MBps. Clearly, in this case, a uniform data placement, where each data source (resp., mapper) pushes an equal amount of data to each mapper (resp., reducer), produces the minimum makespan.

Now, consider a slight modification to the above scenario. Assume now that the non-local links become much slower and can only transmit at 10 MBps, while all other parameters remain the same. Now, a uniform data placement no longer produces the best makespan. Since the non-local links are much slower, it makes sense to avoid non-local links when possible. In fact, a plan where each data source pushes all of its data to its own local mapper completes the push phase in 150 GB/100 MBps = 1500 seconds, while the uniform push would have taken 75 GB/10 MBps = 7500 seconds. Although the map phase for the uniform placement would be faster by 50 GB/100 MBps = 500 seconds, the local-push approach makes up for this through a more efficient data push.

Finally, consider the same network parameters above where local links are fast (100 MBps) and non-local links are slow (10 MBps). Imagine that α, however, is equal to 10, implying that there is 10 times more data in the shuffle phase than in the push phase. The local push no longer performs well, since it does nothing to alleviate the communication bottleneck in the shuffle phase. To avoid non-local communication in the shuffle phase, it makes sense for data source D_2 to push all of its 50 GB of data to mapper M_1 in the push phase, so that all the reduce can happen within cluster 1 without having to use non-local links in the communication-heavy shuffle phase. This is an example of how an optimization would have to look at *all* phases in an *end-to-end* fashion to find a better makespan. In fact, the local push still minimizes the push time from a *myopic* perspective; i.e., it makes a locally optimal decision which is globally suboptimal in terms of makespan.

2.1.2 Research Contributions

Our work makes several research contributions. First, we develop a framework for modeling the performance of MapReduce in a geographically distributed setting. We use this model to formulate an optimization problem whose solution is an *optimal execution plan* describing the best placement of data and computation within a MapReduce job. Our modeling framework is flexible enough to capture a large number of design choices and optimizations. Our optimization minimizes *end-to-end* makespan and controls multiple phases of execution, unlike existing approaches which may optimize in a *myopic* manner or control only a single phase. Further, optimizations using our model are efficiently solvable as Mixed Integer Programs (MIP) using powerful solver libraries.

Second, we modify Hadoop to implement our optimization outputs and use this modified Hadoop implementation along with network and node measurements from the PlanetLab [14] testbed to validate our model and evaluate our proposed optimization. Our model results show that for a geographically distributed compute environment, our end-to-end, multi-phase optimization can provide nearly 82% and 64% reduction in execution time over myopic and the best single-phase optimizations, respectively. Further, using three different applications over an 8-node testbed emulating PlanetLab network characteristics, we show that our statically optimized Hadoop execution plan achieves 31–41% reduction in runtime over an unmodified Hadoop execution employing

its dynamic scheduling techniques.

Finally, our model-driven optimization provides several insights into the choice of techniques and execution parameters based on application and platform characteristics. For instance, we find that an application's data expansion factor α can influence the optimal execution plan significantly, both in terms of which phases of execution are impacted more, as well as where pipeline parallelism is more helpful. Our results also show that as the network becomes more distributed and heterogeneous, our optimization provides higher benefits (82% for globally distributed sites vs. 37% for a single local cluster over myopic optimization) as it can exploit heterogeneity opportunities better. Further, these benefits are obtained independent of application characteristics.

2.2 Model and Optimization

2.2.1 Model

We successively model the distributed platform, the MapReduce application, valid execution plans, and their makespan.

Distributed Platform

We model the distributed platform available for executing the MapReduce application as a tripartite graph with a vertex set of $V = S \cup M \cup R$, where S is the set of data sources, M is the set of mappers, and R is the set of reducers (See Figure 2.2). The edge set E of the tripartite graph is the complete set of edges $(S \times M) \cup (M \times R)$. Each *node* represents a physical resource; either a data source providing inputs, or a computation resource available for executing the map or reduce computational processes. A node can therefore represent a single physical machine or even a single map or reduce *slot* in a small Hadoop cluster, or it can represent an entire rack, cluster, or data center in a much larger deployment. Each edge represents the communication path connecting a pair of such nodes. The *capacity of a node* $i \in M \cup R$ is denoted by C_i (in bits/second), and captures the computational resources available at that node in units of bits of incoming data that it can process per second. Note that C_i is also application-dependent as different MapReduce applications are likely to require different amounts of computing

resources to process the same amount of data. Likewise, the *capacity of an edge* $(i, j) \in E$ is denoted by B_{ij} and represents the bandwidth (in bits/second) that can be sustained on the communication link that the edge represents.

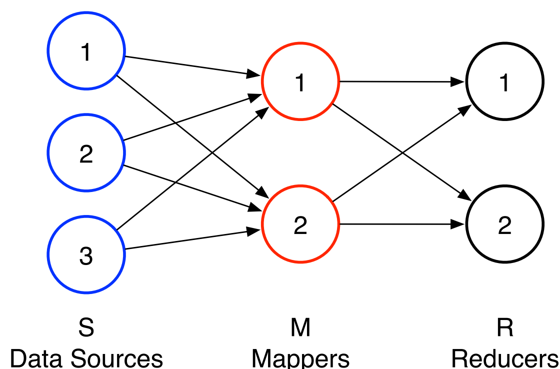


Figure 2.2: A tripartite graph model for distributed MapReduce with 3 data sources, 2 mappers, and 2 reducers.

MapReduce Application

Rather than model the MapReduce application in detail, we model two key parameters. First, we model the *amount of data* D_i (in bits) that originates at data source i, for each $i \in S$. Further, we model an *expansion factor* α that represents the ratio of the size of the output of a map process to the size of its input. Note that α can take values less than, greater than, or equal to 1 depending on whether the output of the map operation is smaller than, larger than, or equal in size to the input, respectively. Many applications perform extensive aggregation in the map phase, for example to count the number of occurrences of each unique object, or to project only a subset of the fields of complex records. These applications have α much less than 1. On the other hand, some applications may require intermediate data to be broadcast from one mapper to multiple reducers [25], or to otherwise transform the inputs to produce larger intermediate data, yielding $\alpha > 1$. The value of α can be determined by profiling the MapReduce application on a sample of inputs [26, 27].

Valid Execution Plans and their Makespan

We define the notion of an *execution plan* to capture the manner in which data and computation of a MapReduce application are scheduled on a distributed platform. Intuitively, an execution plan determines how the input data are distributed from the sources to the mappers and how the intermediate key-value pairs produced by the mappers are distributed across the reducers. Thus, the execution plan determines which nodes and which communication links are used and to what degree, and is therefore a major determinant of how long the MapReduce application takes to complete.

An execution plan of a MapReduce application on a distributed platform is represented by variables x_{ij}, for $(i, j) \in E$, that represent the fraction of the outgoing data from node i that is sent to node j.

Valid Execution Plans. We now mathematically express sufficient conditions for an execution plan to be valid and implementable in a MapReduce software framework while obeying the MapReduce application semantics.

$$\forall (i,j) \in E : 0 \leq x_{ij} \leq 1 \tag{2.1}$$

$$\forall i \in V : \sum_{(i,j) \in E} x_{ij} = 1 \tag{2.2}$$

Equations 2.1 and 2.2 simply express that for each i, the x_{ij}'s are fractions that sum to 1.

The semantics of a MapReduce application requires that each intermediate key be shuffled to a single reducer. Define variable y_k, $k \in R$, to be the fraction of the key space mapped to reducer k. We enforce the one-reducer-per-key requirement with the following constraint.

$$\forall j \in M, k \in R : x_{jk} = y_k \ . \tag{2.3}$$

We define an execution plan to be *valid* if Equations 2.1, 2.2, and 2.3 hold.

Modeling the Consecutive Execution of Phases. The push, map, shuffle, and reduce phases are executed in sequence. Between each pair of consecutive phases, there are three possible assumptions that we can make. The simplest assumption is that a

global barrier exists between the two consecutive phases, in which case all nodes must complete the current phase before any node can proceed to the next. While the global barrier satisfies data dependencies across the different phases, it limits parallelism.

Alternately, one can assume that a *local barrier* exists between the two consecutive phases. With a local barrier, each node can move onto the next phase as soon as it finishes the current phase, without regard to the progress of other nodes. This allows a greater degree of parallelism between the execution of the different phases, allowing the makespan to be reduced.

The third option of *pipelining* provides the greatest amount of parallelism and has the potential for the lowest makespan among the three options. Pipelining allows a node to start the map or reduce phase as long as the *first piece* of its input data is available for processing; i.e., the node need not wait for all of its inputs to be present but rather it can start as soon as it receives the first piece.

Which of these three options is allowable for each pair of consecutive phases depends on the application. For instance, if the application allows incremental processing in each phase without requiring all input data to be received first, then pipelining is allowable. If, on the other hand, the reduce computation cannot begin until it has received all of its intermediate data, then either a local or global barrier at the shuffle/reduce boundary is required.

We now show how to model each of these three possibilities starting with the simplest case where all barriers are global.

Makespan of a Valid Execution Plan with Global Barriers. Given a valid execution plan for a MapReduce application, we now model the total time to completion, i.e., its makespan. To model the makespan, we successively model the time to completion of each of the four phases assuming that a global barrier exists after each phase. We assume that the data is available at all the data sources at time zero when the push phase begins. For each mapper $j \in M$, the time for the push phase to end is denoted by push_end_j and equals the time when all its data is received; i.e.,

$$\forall j \in M : \text{push_end}_j = \max_{i \in S} \frac{D_i x_{ij}}{B_{ij}} \ . \tag{2.4}$$

Since we assume a global barrier after the push phase, all mappers must receive their data before the push phase ends and the map phase begins. Thus, the time when the map phase starts, denoted by map_start, obeys the following equation.

$$\text{map_start} = \max_{j \in M} \text{push_end}_j \ . \tag{2.5}$$

The computation at each mapper takes time proportional to the data pushed to that mapper. Thus, the time map_end_j for mapper $j \in M$ to complete its computation obeys the following.

$$\forall j \in M : \text{map_end}_j = \text{map_start} + \frac{\sum_{i \in S} D_i x_{ij}}{C_j} \ . \tag{2.6}$$

As a result of the global barrier, the shuffle phase begins when all mappers have finished their respective computations. Thus, the time shuffle_start when the shuffle phase starts obeys the following.

$$\text{shuffle_start} = \max_{j \in M} \text{map_end}_j \ . \tag{2.7}$$

The shuffle time for each reducer is governed by the slowest shuffle link into that reducer. Thus the time when the shuffle phase ends at reducer $k \in R$, denoted by shuffle_end_k, obeys the following.

$$\forall k \in R : \text{shuffle_end}_k = \text{shuffle_start} + \max_{j \in M} \frac{\alpha \sum_{i \in S} D_i x_{ij} x_{jk}}{B_{jk}} \ . \tag{2.8}$$

Following the global barrier assumption, the reduce phase computation begins only after all reducers have received all of their data. Let reduce_start be the time when the reduce phase starts. Then

$$\text{reduce_start} = \max_{k \in R} \text{shuffle_end}_k \ . \tag{2.9}$$

Reduce computation at a given node takes time proportional to the amount of data shuffled to that node. Thus, the time when the reduce phase ends at node k, denoted by reduce_end_k, obeys the following.

$$\forall k \in R : \text{reduce_end}_k = \text{reduce_start} + \frac{\alpha \sum_{i \in S} \sum_{j \in M} D_i x_{ij} x_{jk}}{C_k}. \tag{2.10}$$

Finally, makespan equals the time at which the last reducer finishes. Thus

$$\text{Makespan} = \max_{k \in R} \text{reduce_end}_k \ . \tag{2.11}$$

Modeling Local Barriers and Pipelining. We now show how to modify the above constraints to model local barriers and pipelining. First, we remove the constraints defining the start time for the last three phases as expressed in Equations 2.5, 2.7 and 2.9. Then we generalize the definitions for the ending time of a stage by defining a combination operator $\oplus$ for each optimization as follows:

$$a \oplus b = \begin{cases} a + b, & \text{if local barrier} \\ \max(a, b), & \text{if pipelined} \end{cases}$$

Then we replace the constraints in Equations 2.6, 2.8, and 2.10 with new constraints that capture the fact that a node can start its next phase without waiting for all nodes to complete the previous phase. For the ending time of the map phase, Equation 2.6 is replaced with

$$\forall j \in M : \text{map_end}_j = \text{push_end}_j \oplus \frac{\sum_{i \in S} D_i x_{ij}}{C_j} \ . \tag{2.12}$$

Intuitively, the above constraint specifies that the runtime for the map phase on a node depends on the end time of the push phase on that node as well as the time for map computation on all of the data pushed to that node. For the local barrier case, this equals the sum of the two times (corresponding to the time to push and then compute in sequence). On the other hand, for the pipelining case, this equals the maximum of the two times, since the map phase at a node cannot end until all of its data has arrived and all of its computation has finished, and the slower of these two operations will dictate the completion time. Note that we are assuming that the data push and map computation are completely overlapped. This assumption is valid if the total quantity of data is much larger than individual record size, which is the case for many typical MapReduce applications.

Based on similar intuition, the constraint for the shuffle stage in Equation 2.8 is replaced with

$$\forall k \in R : \text{shuffle_end}_k = \max_{j \in M} \left\{ \text{map_end}_j \oplus \frac{\alpha \sum_{i \in S} D_i x_{ij} x_{jk}}{B_{jk}} \right\} . \tag{2.13}$$

Finally, the constraint for the end of the reduce stage in Equation 2.10 is replaced with

$$\forall k \in R : \text{reduce_end}_k = \text{shuffle_end}_k \oplus \frac{\alpha \sum_{i \in S} \sum_{j \in M} D_i x_{ij} x_{jk}}{C_k}. \quad (2.14)$$

Discussion of Assumptions. We have made some assumptions in our formulation. For example, we have not constrained the amount of data pushed or shuffled to any given node, or the amount of intermediate data stored at a node. Extending our model to include such constraints would be straightforward.

We have also assumed that network links are independent based on the assumption that a MapReduce job would be unable to saturate highly multiplexed wide-area links. It is, however, possible to extend the model to account for the fact that some logical links actually share a physical link. For example, to reflect that logical links in the push and shuffle phases share the same physical link, we could add a per-shared-link bandwidth constraint.

Additionally, we have assumed that the time for map and reduce computation is linear in the amount of data processed. If a particular application deviates significantly from this assumption, then a nonlinear computation time can be modeled by a piecewise linear approximation. Such an approximation would take the place of Equations 2.6, 2.10, 2.12, or 2.14 as appropriate.

2.2.2 Model Validation

To validate our model, we modify the Hadoop MapReduce framework to follow specific execution plans. We show how our model predicts makespan given the execution plan and network and compute rates as input.

Implementing an Execution Plan in Hadoop

We modify two key aspects of Hadoop, specifically how map and reduce tasks are *defined* and how they are *scheduled*. We discuss these aspects first for push and map, then for shuffle and reduce.

First, consider how map tasks are defined. Just as in vanilla (i.e., unmodified) Hadoop, we define map tasks that each represent a small partition of the input data;

no more than 64 MB in our experiments. But unlike vanilla Hadoop, where each of these partitions typically corresponds to a contiguous segment of a file stored on a particular host (i.e., an HDFS file block), our partitions combine inputs from several data sources. For example, if the execution plan calls for mapper M to receive $\frac{3}{4}$ of its input from source S_1, and $\frac{1}{4}$ from source S_2, then a 64 MB partition created for mapper M will consist of 48 MB from S_1 and 16 MB from S_2. To encode this in a form usable by Hadoop, we implement the `InputFormat` interface. This involves implementing an `InputSplit` to describe the task inputs as well as a `RecordReader` to provide map tasks a means for reading input key-value pairs. Our `RecordReader` opens TCP sockets to multiple data sources, which we have implemented as simple TCP servers, and uses each of these sockets to read the amount of data described in its `InputSplit`.

Now consider scheduling these map tasks in Hadoop. Each task is intended to run on a specific host, and we include this intended host as a field of the `InputSplit` object. When scheduling a map task, Hadoop queries the `InputSplit` for that task to determine where it would be considered local. Then Hadoop makes its best effort to assign the task to one of these local nodes. To strictly enforce our execution plans, however, this best-effort basis is not enough; we modify Hadoop to assign tasks exactly where we request them. Specifically, we make minor modifications to Hadoop's `JobInProgress` class, which is responsible for returning new map tasks to the task scheduler on request. These changes force `JobInProgress` to return only local tasks when a new configuration option (call it `LocalOnly` for brevity) is set.

Next, consider task definition for the reduce phase. Each reduce task in Hadoop corresponds to a partition of the intermediate key space. It is the responsibility of the `Partitioner` to determine the partition for each intermediate key-value pair. By default, Hadoop uses a hash function for this purpose: It assigns an intermediate key-value pair to partition $(h \mod p)$ where h is the key's hash and p is the total number of partitions.[1] We make the assumption that the `Partitioner` creates roughly equal-sized partitions. To implement our execution plan, however, we need to create arbitrarily large or small partitions. For example, since we assume that each reduce node will run the same number of tasks, a reduce node k with a higher y_k (Equation 2.3) will need

[1] It is also common for Hadoop users to define their own `Partitioner`, for example to implement a relational join, or to use range partitioning instead of hash partitioning.

larger partitions than one with a smaller y_k. To achieve this non-uniformity in partition sizes, we add a level of indirection: The user's `Partitioner` is first used to create a number of smaller partitions, then ranges of these small sub-partitions are combined into the final reduce partitions. Concretely, consider a case with two reducers R_1 and R_2 where the plan calls for R_1 to reduce $\frac{3}{4}$ of the intermediate key-value pairs. Keys are partitioned into say 16 sub-partitions, and the first 12 of these are combined into R_1's partition, while the last 4 are combined into R_2's partition.

Finally, consider how these reduce tasks are scheduled. In vanilla Hadoop, reduce tasks are assigned to arbitrary hosts on demand; there is no concept of reduce task locality in vanilla Hadoop. In our modified version, we establish a mapping between reduce hosts and partitions before the job begins. We modify `JobInProgress` to assign only specific partition numbers to a requesting reduce host.

Instantiating Barrier Configurations

Our model is general enough to capture different barrier configurations (global, local and pipelining) at each phase boundary of a MapReduce job. We instantiate a subset of all possible barrier configurations; Hadoop supports some of these by default, while we do not consider some of the others that are either hard to implement or are not immediately interesting.

Push/map: To achieve a global barrier at the push/map phase boundary, we run a separate map-only job to enact the push. This job uses the custom `InputFormat` to read directly from data sources, and writes directly to HDFS. Then we run the main job, which uses a regular `FileInputFormat` to read directly from HDFS as a typical Hadoop MapReduce job would. This is the same way the `DistCP` tool from the Hadoop distribution copies files from one distributed file system to another. We achieve pipelining by using the custom `InputFormat` in the main job itself to read directly from the data sources to the mappers. We do not instantiate a local barrier at this phase boundary.

Map/shuffle: For global barriers, we set the Hadoop configuration parameter `mapred.reduce.slowstart.completed.maps` to 1. This parameter specifies the fraction of map tasks that must complete before any reduce task is scheduled. Since

the shuffle is actually carried out by reducers pulling their data, if no reducers start until all map tasks finish, then there is also no shuffle until the map phase finishes; i.e., the map and shuffle phases are separated by a global barrier. Coarse-grained pipelining is essentially the default behavior of Hadoop as long as there are enough reduce slots to schedule all reduce tasks immediately. MapReduce Online [28] implements finer-grained pipelining between these phases, but we consider only Hadoop's coarse-grained pipelining here. We do not instantiate a local barrier at this phase boundary.

Shuffle/reduce: Here, a local barrier is the default configuration of Hadoop, as a reducer can start as soon as it has finished receiving all of its input data. Implementing pipelining at this boundary is difficult in general: The MapReduce programming model requires that each invocation of the reduce function be provided an intermediate key and *all* intermediate values for that key. Relaxing the barrier at this phase therefore requires changes at the application level, as Verma et al. [29] describe in detail. We do not implement a global barrier at this phase boundary.

Parameter Estimation

Next, we show how the various parameters of our model can be estimated. We use PlanetLab [14] for our measurements, since it is a globally distributed testbed and is representative of the geographically distributed environments that we consider in this chapter. We measure bandwidth and compute capacities for a set of PlanetLab nodes, and estimate the model parameters for our distributed experimental platform as follows. For each $(i, j) \in E$, bandwidth B_{ij} is estimated by transferring data over that link and measuring the achieved bandwidth. For stable estimates, we used downloads of size at least 64 MB or a transfer time of at least 60 seconds, whichever occurs first. For each compute node i, we estimate its compute rate C_i by measuring the runtime of a simple computation over a list within a Python program, yielding a rate of computation in MBps. This value is not directly useful on its own, but we use it to estimate the *relative* speeds of the compute nodes.

We base these measurements on a set of eight PlanetLab nodes distributed across

eight sites, including four in the United States, two in Europe, and two in Asia. The unscaled C_i values on these nodes range from as low as 9 MBps to as high as about 90 MBps. Table 2.1 shows the intra-continental and inter-continental average link bandwidths for these nodes and highlights the heterogeneity that characterizes such a geographically distributed network.

Table 2.1: Measured bandwidth (KBps) of the slowest/fastest links between nodes in each continent.

	US	EU	Asia
US	216 / 9405	110 / 2267	61 / 3305
EU	794 / 2734	4475 / 11053	1502 / 1593
Asia	401 / 3610	290 / 1071	23762 / 23875

Compute and Network Emulation

We use an emulated local testbed to achieve stable and repeatable experiments, and to avoid the severe memory constraints on PlanetLab, which prevent such data-intensive experiments. We emulate the full heterogeneity of the PlanetLab environment by using the `tc` utility to emulate network heterogeneity, and a synthetic Hadoop MapReduce job to emulate computation heterogeneity and the application parameter α. Mappers in this synthetic job read a key-value pair and emit that same key-value pair an appropriate number of times to achieve the user-specified α value. For example, with $\alpha = 0.5$, the synthetic mapper directly emits only every other input key-value pair; with $\alpha = 2$, it emits each input key-value pair twice. This job uses an identity reducer. This synthetic application also allows us to emulate computation heterogeneity by scaling the amount of computation on each node based on a user-provided parameter.

We run this synthetic Hadoop MapReduce job using our modified Hadoop implementation on our local cluster of eight nodes. Each node has two dual-core 2800 MHz AMD Opteron 2200 Processors, 4 GB RAM, a 250 GB disk, and Linux kernel version 2.6.32. The nodes are connected via Gigabit Ethernet, though we use `tc` to emulate the network bandwidths measured on PlanetLab.

Validation Results

We study the predictive power of our model by correlating its predictions with measured makespan using the following approach. First, we need to scale the C_i values, which we do by running the same profiling code on our local cluster as we did on the PlanetLab nodes. The unscaled C_i values are then multiplied by a constant factor such that the greatest C_i is equal to that measured on our local cluster. Then, given these scaled C_i values along with the bandwidth measurements from PlanetLab, we derive the model's prediction for makespan by using the model equations (Equations 2.1 to 2.11, with substitutions as needed for local barriers or pipelining).

For our validation, we consider α values of 0.1, 1, and 2. We consider two levels of network heterogeneity: the network conditions measured from PlanetLab as well as no emulation (raw LAN bandwidths). We also consider two levels of computation heterogeneity: those measured from PlanetLab, as well as no emulation. We consider all combinations of global barriers and pipelining at the push/map phase boundary and the map/shuffle boundary. We use a local barrier at the shuffle/reduce boundary, leading to a total of four barrier configurations. We include two types of execution plans: a uniform plan that distributes the input and intermediate data uniformly among the mappers and reducers respectively, as well as an optimized plan generated by our optimization algorithm (see Section 2.2.3). All of our plans use 256 MB of input data from each of the eight data sources. We use two mappers and one reducer on each of eight compute nodes.

The results of the validation are shown in Figure 2.3, where we observe a strong correlation (R^2 value of 0.9412) between predicted makespan and measured makespan. In addition, the linear fit to the data points has a slope of 1.1464, which shows there is also a strong correspondence between the absolute values of the predicted and measured makespan.

2.2.3 Optimization Algorithm

Having defined and validated our model, we now use it to find the execution plan that minimizes the makespan of a MapReduce job. To do so, we formulate an optimization problem that has two key properties. First, it minimizes the *end-to-end* makespan of

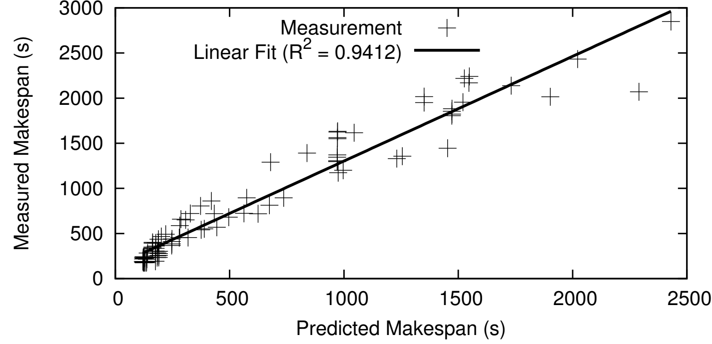


Figure 2.3: Measured vs. model-predicted makespan from the Hadoop implementation on the local emulated testbed.

the MapReduce job, including the time from the initial data push to the final reducer execution. Thus, its decisions may be suboptimal for individual phases, but will be optimal for the overall execution of the job. Second, our optimization is *multi-phase* in that it controls the data dissemination across both the push and shuffle phases; i.e., it outputs the best way to disseminate both the input and intermediate data so as to minimize makespan.

Viewing Equations 2.1 to 2.11 as constraints, we need to minimize an objective function that equals the variable Makespan. (If the barriers are not global, the appropriate substitutions of the constraints need to be made.) To perform this optimization efficiently, we rewrite the constraints in linear form to obtain a Mixed Integer Program (MIP). Writing it as MIP opens up the possibility of using powerful off-the-shelf solvers such as the Gurobi version 5.0.0 that we use to derive our results.

There are two types of nonlinearities that occur in our constraints, and these need to be converted to linear form. The first type is the existence of the max operator. Here we use a standard technique of converting an expression of the form $\max_i z_i = Z$ into an equivalent set of linear constraints $\forall i : z_i \leq Z$, where Z is minimized as a part of the overall optimization.

The second type of non-linearity arises from the quadratic terms in Equations 2.8 and 2.10 (or Equations 2.13 and 2.14 as appropriate). We remove this type of nonlinearity with a two step process. First, we express the product terms of the form yz that involve two different variables in separable form. Specifically, we introduce two new variables

$w = \frac{1}{2}(y+z)$ and $w' = \frac{1}{2}(y-z)$. This allows us to replace each occurrence of yz with $w^2 - w'^2$, which is in separable form since it does not involve a product of two different variables. Next, we approximate the quadratic terms w^2 and $-w'^2$ with a piecewise linear approximation of the respective function. The more piecewise segments we use, the more accuracy we achieve, but this requires a larger number of choice variables resulting in a longer time to solve. As a compromise, we choose 10 evenly spaced points on the curve leading to an approximation with 9 line segments resulting in a worst-case deviation of 4.15% between the piece-wise linear approximation and the actual function. Since w^2 is a convex function, its piece-wise linear approximation can be expressed using linear constraints with no integral variables. However, since $-w^2$ is not a convex function, its piecewise linear approximation requires us to utilize new binary integral variables to choose the appropriate line segment, resulting in a MIP (Mixed Integer Program) instead of an LP (Linear Program).[2]

2.3 Model and Optimization Results

To understand the benefits of our end-to-end multi-phase optimization relative to other schemes, we use our model to perform several comparisons.

2.3.1 Modeled Environments

Before we present the results of these comparisons, we describe the environments that we consider. We use PlanetLab measurements to create the different distributed network environments that we use throughout this section. Our network environments vary from a single data center that is relatively homogeneous, to a diverse environment comprising eight globally distributed data centers. To realistically model these environments, we use actual measurements of the compute speeds and link bandwidths from PlanetLab nodes distributed around the world, including four in the US, two in Europe, and two in Asia with compute rates and interconnection bandwidths as described in Section 2.2.2. Using these measurements, we generate the following network environments:

Local data center: This setup consists of eight nodes in a single data center in the

[2] For a more detailed treatment of the techniques that we have used to remove the two types of nonlinearities, see [30].

US, with each node hosting a source, mapper, and reducer. This setup corresponds to the traditional local MapReduce execution scenario.

Intra-continental data centers: This setup consists of two data centers located within a continent: All nodes are in the US, specifically in Texas or California. This setup corresponds to a more distributed topology than the local data center scenario.

Globally distributed data centers: In this setup, nodes span the entire globe, introducing much greater heterogeneity. We consider two global environments: one with four data centers (in California, Texas, Germany, and Japan), and another with eight data centers (same as earlier with additional nodes in California, Illinois, the UK, and Japan). This allows us to study the impact of scaling up the number of locations.

For each of the above setups, the total number of nodes is held constant at eight. In some cases, where we did not have sufficiently many actual nodes to meet this requirement (e.g., for the local data center setup), we added virtual replica nodes to the setup with the measured node/link characteristics of the corresponding real nodes. In addition, we held the number of data sources fixed, allocating these data sources to data centers in the same proportion as mappers and reducers. The total amount of input data per data source was held constant throughout.

The globally distributed environment with eight data centers is most appropriate for studying the general properties of our optimization, as it closely resembles the geographically distributed settings that we focus on in our work. Consequently, we use this environment in all our experiments, except in Section 2.3.6 where we study the impact of distribution of resources on our optimization, and therefore use all of the above environments.

2.3.2 Heuristics vs. Model-Driven Optimization

We compare our optimal execution plans to several baseline heuristics, which we refer to as *uniform*, *best-centralized* and *push-to-local*. In the uniform heuristic, sources push uniformly to all mappers, and mappers shuffle uniformly to all reducers. This reflects

the expected distribution if data sources and mappers were to randomly choose where to send data. We model this heuristic by adding the following additional constraints.

$$\text{Uniform Push}: \quad \forall i \in S, j \in M : x_{ij} = \tfrac{1}{|M|} \tag{2.15}$$

$$\text{Uniform Shuffle}: \quad \forall j \in M, k \in R : x_{jk} = \tfrac{1}{|R|} \tag{2.16}$$

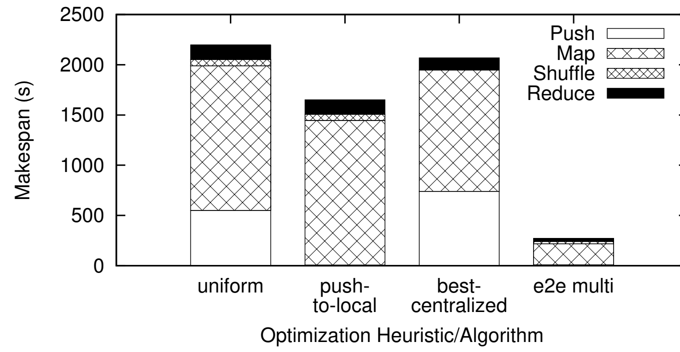
Best-centralized pushes data to a single mapper node and performs all map computation there. Likewise, all reduce computation is performed at a single reducer node at the same location. We can model this heuristic by simply excluding all mappers and reducers other than the selected centralized nodes, and then using the model as usual. We compute the makespan for each possible choice of centralized location, and report the minimum among these values, hence the name best-centralized.

Push-to-local pushes data from each source to the most local mapper (in this case, to a mapper on the same node) and uses a uniform shuffle; it is analogous to the purely distributed extreme. We model this heuristic by adding the uniform shuffle constraint from Equation 2.16 as well as a new local push constraint:

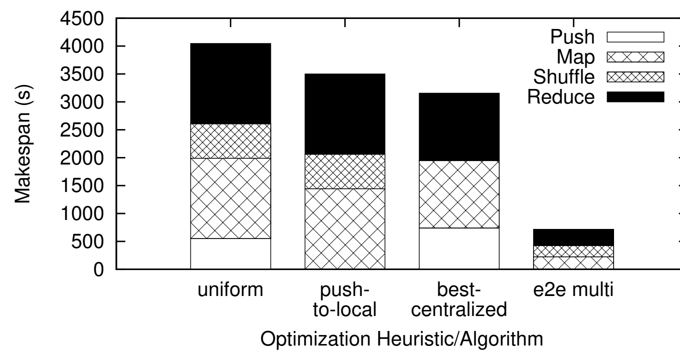
$$\forall i \in S, j \in M : x_{ij} = \begin{cases} 1 & \text{if } j \text{ is the mapper closest to source } i \\ 0 & \text{otherwise} \end{cases} \tag{2.17}$$

Figure 2.4 shows the makespan achieved by each of these heuristics, as well as by our end-to-end multi-phase optimization algorithm (e2e multi) for three values of α. We make several key observations from these results. First, both the uniform and best-centralized heuristics have poor push performance relative to the push-to-local heuristic and our end-to-end optimization. The reason is that the uniform and best-centralized heuristics place heavy traffic on slow wide-area links while the push-to-local heuristic uses fast local links for the push. Our optimization approach intelligently decides how to utilize links to minimize end-to-end makespan.

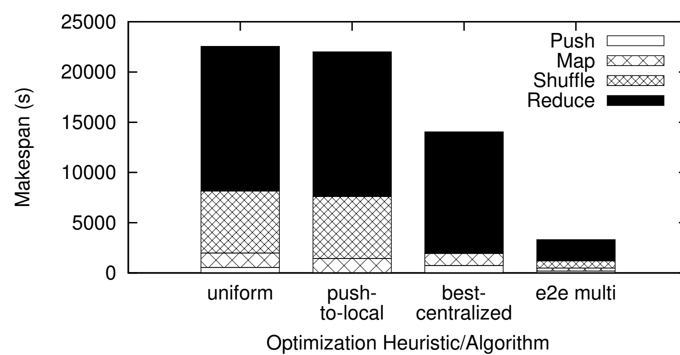
Second, we see that the best-centralized heuristic becomes more favorable relative to the push-to-local heuristic as α increases, demonstrating that neither is universally better than the other. The reason is that, as α increases, the shuffle phase becomes heavier relative to the push phase. The best-centralized heuristic avoids wide-area links in the shuffle, and this becomes increasingly valuable as the shuffle becomes dominant.



(a) α=0.1



(b) α=1



(c) α=10

Figure 2.4: A comparison of three optimization heuristics and our end-to-end multi-phase optimization (e2e multi).

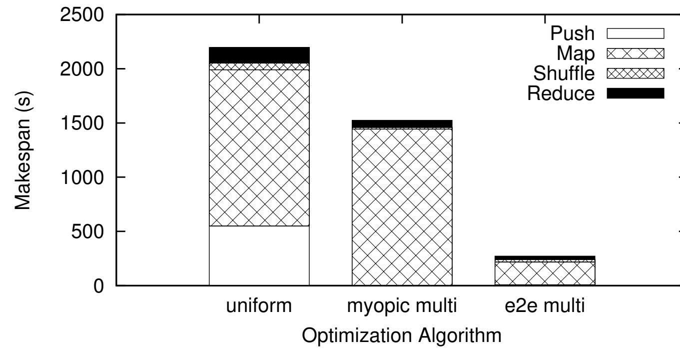
Finally, we see that our end-to-end multi-phase optimization approach is significantly better than any of the three heuristics, reducing makespan over the best heuristic by 83.5%, 77.3%, and 76.4% for α=0.1, 1, and 10 respectively. This demonstrates that the best data placement is one that considers resource and application characteristics.

2.3.3 End-to-End vs. Myopic

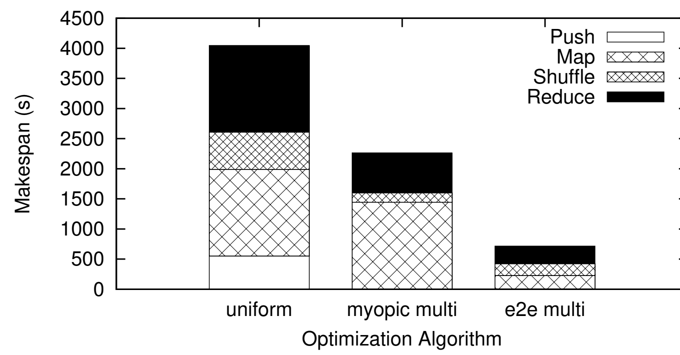
Now we study the benefit of optimizing end-to-end makespan compared to *myopically* minimizing the time for a single phase. The distinction between these two alternatives is the *objective function* that is being minimized. With end-to-end, the objective function is the overall makespan of the MapReduce job, whereas a myopic optimizer uses the time for single data dissemination phase (push or shuffle) as its objective function. Myopic optimization is a localized optimization, e.g., pushing data from data sources to mappers in a manner that minimizes the data push time. Such local optimization might result in suboptimal global execution times by creating bottlenecks in other phases of execution. Note that myopic optimization can be applied to both push and shuffle phases in succession. That is, the push phase is first optimized to minimize push time and then the shuffle phase is optimized to minimize shuffle time assuming that the input data has been pushed to mappers according to the first optimization.

The myopic optimizations described above can be achieved by modifying our formulation in Section 2.2 as follows. Instead of minimizing makespan, we replace Equation 2.11 with alternate objective functions. For optimizing the push time we minimize $\max_{j \in M} \text{push_end}_j$, and to optimize the shuffle time we minimize $\max_{k \in R} \text{shuffle_end}_k$. (Computing a myopic multi-phase plan requires solving multiple optimizations in sequence.)

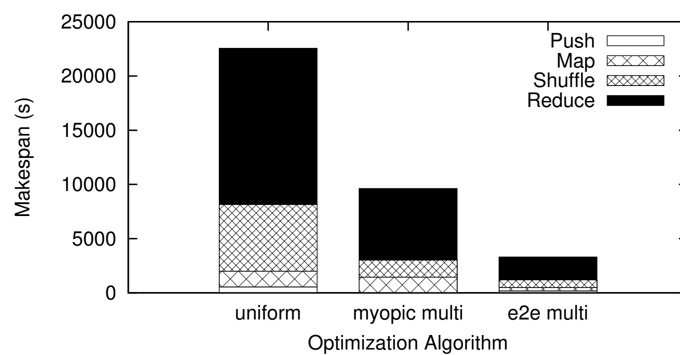
In Figure 2.5, we show the makespan achieved in three different cases: (i) for a uniform data placement; (ii) for a myopic, multi-phase optimization, where the push and shuffle phases are optimized myopically in succession; and (iii) for our end-to-end, multi-phase optimization that minimizes the total job makespan. Note that since both (ii) and (iii) are multi-phase, the primary difference between them is that one is myopic and the other is end-to-end, helping us determine the relative merits of end-to-end versus myopic approaches. We evaluate these three schemes for different assumptions for α. We see that for each α, the myopic optimization reduces the makespan over the uniform data



(a) α=0.1



(b) α=1



(c) α=10

Figure 2.5: Our end-to-end multi-phase optimization (e2e multi) compared to a uniform heuristic and a myopic optimization approach.

placement approach (by 30%, 44%, and 57% for $\alpha = 0.1$, 1, and 10 respectively), but is significantly outperformed by the end-to-end optimization (which reduces makespan by 87%, 82%, and 85%). This is because, although the myopic approach makes locally optimal decisions at each phase, these decisions may be globally suboptimal. Our end-to-end optimization makes globally optimal decisions. As an example, for α=0.1, while both the myopic and end-to-end approaches dramatically reduce the push time over the uniform approach (by 99.4% and 98.5% respectively), the end-to-end approach is also able to reduce the map time substantially (by 85%) whereas the myopic approach makes no improvement to the map time. A similar trend is evident for α=10, where the end-to-end approach is able to lower the reduce time significantly (by 68%) over the myopic approach. These results show the benefit of an end-to-end, globally optimal approach over a myopic, locally optimal but globally suboptimal approach.

2.3.4 Single-Phase vs. Multi-Phase

We continue by comparing the performance of our multi-phase optimization to that of a single-phase optimization. The distinction between single-phase and multi-phase is which phase (push, shuffle, or both) is *controlled* by the optimization. This is orthogonal to the end-to-end versus myopic distinction. A single-phase optimization controls the data distribution of one phase—e.g., the push phase—alone, while using a fixed data distribution for the other communication phase. A single-phase optimization is myopic if it minimizes the time for that phase alone. However, it could also be end-to-end if it optimizes the phase so as to achieve the minimum overall makespan. We model a single-phase optimization by using one of the uniform push or shuffle constraints (Equation 2.15 or 2.16) to constrain the data placement for one of the phases, while allowing the other phase to be optimized.

In Figure 2.6, we compare (i) a uniform data placement, (ii) an end-to-end single-phase push optimization that assumes a uniform shuffle, (iii) an end-to-end single-phase shuffle optimization that assumes a uniform push, and (iv) our end-to-end multi-phase optimization. Note that both the single-phase optimizations here are end-to-end optimizations in that they attempt to minimize the total makespan of the MapReduce job. The primary difference between (ii) and (iii) on the one hand and (iv) on the other hand is that the former are single-phase and the latter multi-phase, letting us evaluate

the relative benefit of single- versus multi-phase optimization.

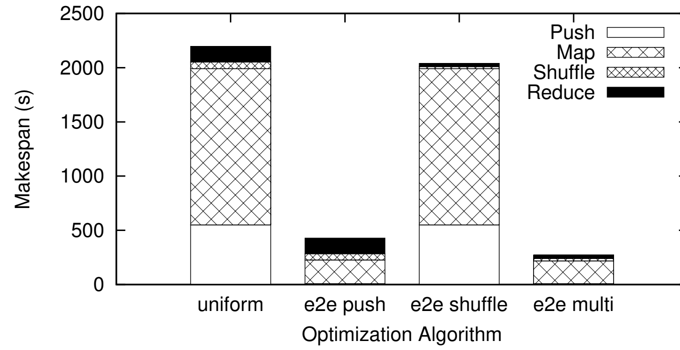
In Figure 2.6, we observe that across all α values, the multi-phase optimization outperforms the best single-phase optimization (by 37%, 64%, and 52% for α=0.1, 1, and 10 respectively). This shows the benefit of controlling the data placement across multiple phases. Further, for each α value, optimizing the bottleneck phase brings greater reduction in makespan than optimizing the non-bottleneck phase. For instance, for α=0.1, the push and map phases dominate the makespan in the baseline (25% and 66% of the total runtime for uniform, respectively) and push optimization alone is able to reduce the makespan over uniform by 80% by lowering the runtime of these two phases. On the other hand, for α=10, the shuffle and reduce phases dominate (27% and 64% of total runtime for uniform, respectively) and optimizing these phases via shuffle optimization reduces makespan by 69% over uniform.

By influencing the data placement across multiple phases, our multi-phase optimization improves both the bottleneck as well as non-bottleneck phases. When there is no prominent bottleneck phase (α=1), the multi-phase optimization outperforms the best single-phase optimization substantially (by 64%). These results show that the multi-phase optimization is able to automatically optimize the execution independent of the application characteristics.

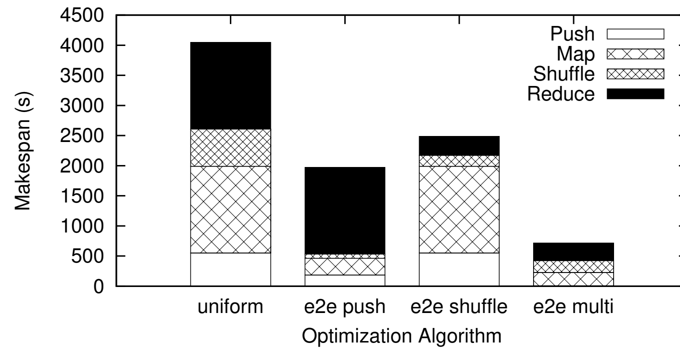
An additional interesting observation from Figures 2.6(b) and (c) is that optimizing earlier phases can have a positive impact on the performance of later phases. In particular, for α=10, push optimization also reduces the shuffle time (by 90%), even though the push and map phases themselves contribute little to the makespan. This is because the location of the mappers to which inputs are pushed has an impact on how intermediate data are shuffled to reducers.

2.3.5 Barriers vs. Pipelining

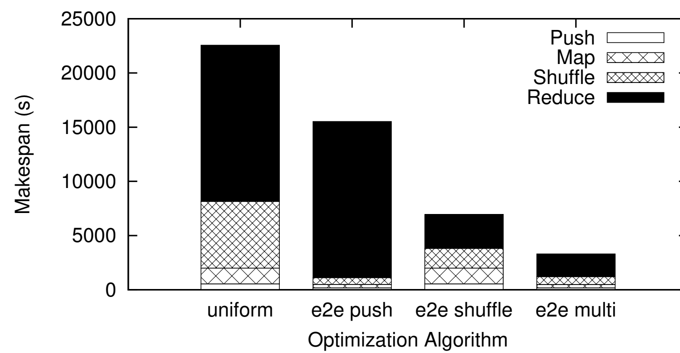
Now we study the impact of relaxing barriers on the makespan predicted by our model. In particular, we focus on the impact of using pipelining vs. global barriers at each phase boundary. In Figure 2.7, we show the normalized makespan for a select set of different barrier configurations. All makespan values are normalized relative to the optimal makespan for a configuration with global barriers at each phase boundary. Each bar shows the effect of relaxing only a single global barrier to pipelining at a time, at



(a) α=0.1



(b) α=1



(c) α=10

Figure 2.6: Our end-to-end multi-phase optimization (e2e multi) compared to a uniform heuristic as well as end-to-end push (e2e push) and end-to-end shuffle (e2e shuffle) optimization.

the push/map, the map/shuffle, and the shuffle/reduce boundaries respectively, as well as the all-pipelined configuration, where all phases are pipelined.

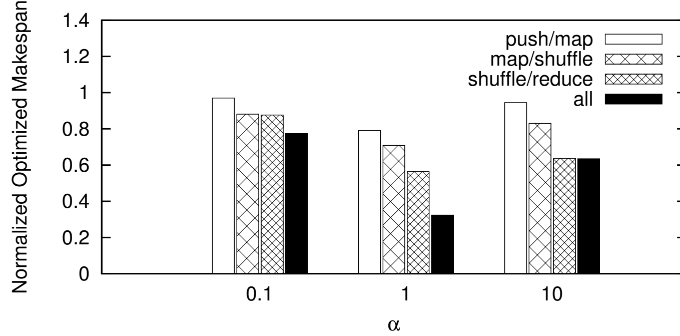


Figure 2.7: A comparison of barrier configurations. Makespan is normalized relative to the optimal makespan for an all-global-barrier configuration. Each bar represents the boundary at which the global barrier is relaxed to pipelining; "all" corresponds to an all-pipelining configuration.

We make two key observations. First, pipelining is most effective when phases are roughly "balanced" in terms of time. The reason is that when balanced phases are overlapped, one can completely hide the latency of the others. When one phase dominates, however, it continues to dominate even when it overlaps shorter phases. For the parameters considered here, the phases are most closely balanced when $\alpha = 1$, as Figure 2.6 shows. Consequently, Figure 2.7 shows that each barrier relaxation provides the greatest benefit when $\alpha = 1$.

Second, relaxing late-stage barriers—such as those between map and shuffle or between shuffle and reduce—is predicted to have a greater benefit than relaxing barriers between push and map stages. The reason is that our optimization is more constrained in data placement during the shuffle phase than the push phase due to the one-reducer-per-key constraint, and hence pipelining finds more remaining opportunity to hide the latency of the shuffle phase with its adjoining computational phases (map or reduce). This phenomenon can also be observed from Figure 2.6, where we see that the shuffle time is higher than the push time with our optimization (e2e multi), particularly for α=1 and 10.

2.3.6 Compute Resource Distribution

To evaluate the effect of distribution of resources, we derive optimized execution plans using our proposed (end-to-end multi-phase) optimization algorithm for *all* of the environments described in Section 2.3.1, starting from a relatively homogeneous environment including a single data center, to an intra-continental setup consisting of two data centers in the US, to globally distributed setups with four and eight data centers.

Figure 2.8 shows the makespan for the myopic optimization that successively optimizes the execution times of the push phase and the shuffle phase, and the end-to-end optimization that optimizes makespan by considering all phases at once. Both the myopic and end-to-end optimizations are compared against a baseline that performs uniform push and shuffle.

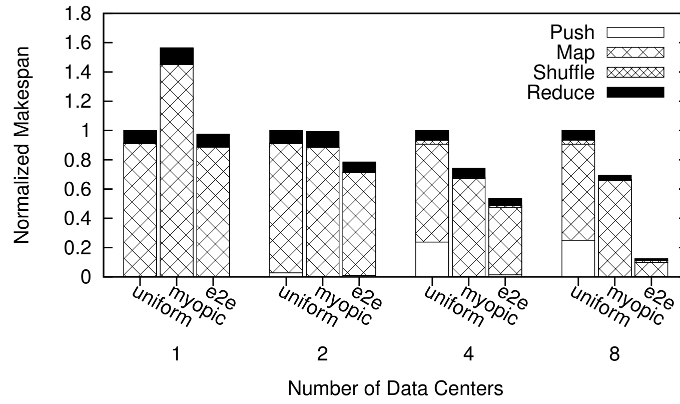
If the network environment is a single data center with relative homogeneity of compute rates and network bandwidths, the uniform baseline does almost as well as end-to-end optimization for all values of α. Interestingly, a uniform heuristic can outperform a myopic optimization in some cases. Intuitively, myopic optimization reacts to rectify small communication imbalances, but this can in turn create larger computational imbalances among the map and reduce tasks, resulting in a longer makespan. This effect can be seen from the figure by the increased map and reduce times for myopic for α=0.1 and 10 respectively.

As the network environment becomes more diverse with more data centers, both myopic and end-to-end optimizations outperform uniform, as uniform fails to account for the diversity of the environment. As expected, end-to-end performs the best with makespans 82–87% lower than uniform and 65–82% lower than myopic.

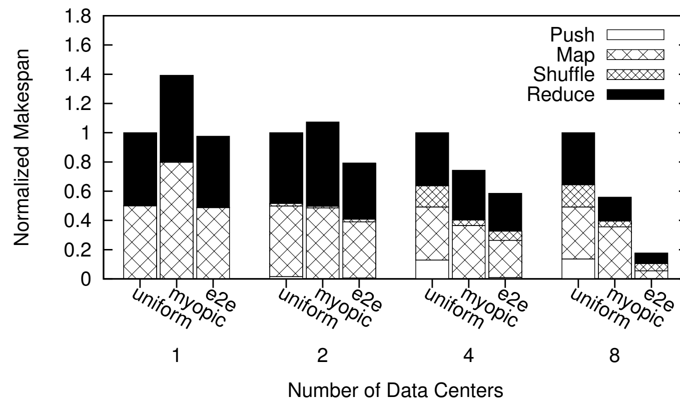
In summary, our results show that our optimization derives much greater opportunity for improvement as the diversity and heterogeneity of the environment increases, while reducing to a largely uniform placement for tightly coupled, homogeneous environments, where myopic optimization may actually hurt performance.

2.3.7 Input Data Distribution

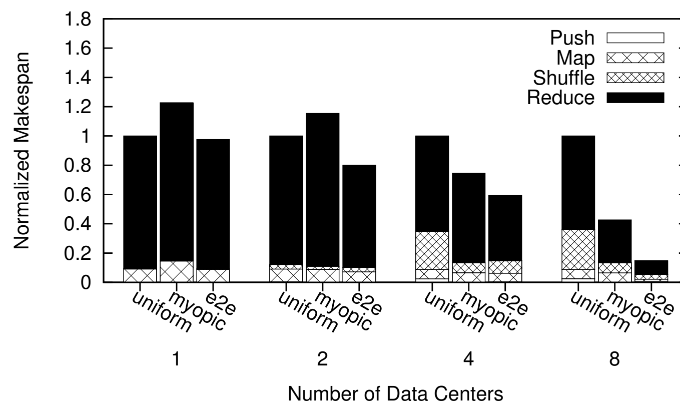
So far we have considered only situations where each data source hosts the same amount of input data. We now explore the effect of a heterogeneous distribution of input data.



(a) α=0.1



(b) α=1



(c) α=10

Figure 2.8: Our end-to-end multi-phase optimization (e2e) compared to a uniform heuristic and myopic optimization for different network environments. Makespan is normalized relative to the uniform baseline.

To do so, we revisit the Akamai log data introduced in Section 1.1, and logically map this data set onto our globally distributed set of PlanetLab data sources as follows. First, we assign each log entry to the nearest of our PlanetLab nodes based on great circle distance. We then count the number of log entries assigned to each PlanetLab node and use this distribution as input to the model. The resulting distribution is highly heterogeneous: The most heavily loaded node supplies 49% of all input data, the top three together supply roughly 80%, and the most lightly loaded supplies less than 1%.

The results for three values of α are shown in Figure 2.9. The overall trends are similar to the homogeneous input distributions in Figure 2.8: A myopic optimization outperforms a uniform heuristic, but our end-to-end multi-phase optimization is much better yet. In fact, for high α, push and map times are small relative to shuffle and reduce times, so the results are quite similar between the homogeneous input distribution and the heterogeneous input distribution. This provides evidence that our optimization approach can improve performance independent of the physical distribution of inputs.

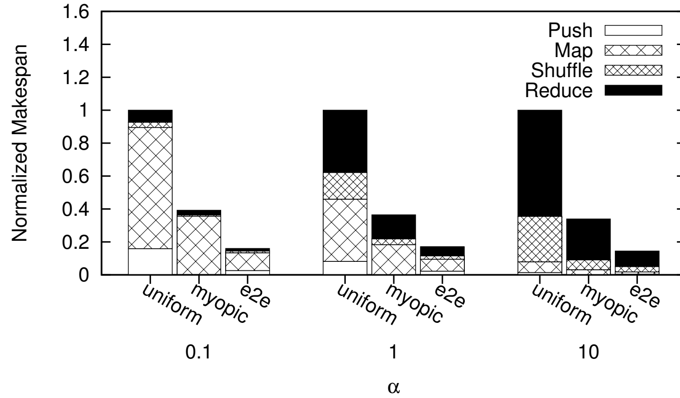


Figure 2.9: Our end-to-end multi-phase optimization (e2e) compared to uniform and myopic baselines in the eight-node PlanetLab environment with inputs distributed according to Akamai logs. Makespan is normalized relative to the uniform baseline.

2.4 Comparison to Hadoop

We compare the results of our optimization against *vanilla Hadoop*, which represents a typical unmodified Hadoop execution.

2.4.1 Experimental Setup

For these experiments, we use an 8-node cluster with bandwidths emulating a distributed PlanetLab environment (Section 2.2.2). We do not emulate compute heterogeneity in this setup; only network links have emulated heterogeneity. Each of our eight physical nodes hosts two map slots and one reduce slot, as well as an HDFS datanode, and a simple TCP server acting as a data source. We use our modified Hadoop implementation (Section 2.2.2), using largely default Hadoop configuration options.

We set the replication factor in HDFS to 1 to prevent replication over emulated wide-area links, as we have observed that wide-area replication adversely affects performance. For example, experiments on vanilla Hadoop using three different applications (see Section 2.4.2) showed that increasing HDFS replication from 1 to 2 can increase average makespan by a factor of 1.4 to 2.1, and increasing replication from 1 to 3 can increase average makespan by a factor of 3.6 to 5.7.[3]

To achieve a fair comparison, we allow vanilla Hadoop to use its dynamic load balancing mechanisms such as speculative task execution and non-local work stealing to mitigate stragglers and avoid idle resources. Additionally, we allow vanilla Hadoop to take advantage of its existing data-locality optimization by hinting (through the `InputSplit`) that each data source should prefer to push to its most local compute node. For our optimization, we derive execution plans based on the underlying platform and application parameters, including the barrier configuration (global, pipelined, and local at the push/map, map/shuffle, and shuffle/reduce boundaries, respectively). To ensure that Hadoop strictly follows our optimized plans, we disable its dynamic mechanisms by enabling our `LocalOnly` option (Section 2.2.2). As a baseline, we also compare vanilla Hadoop and our optimization to a uniform execution plan.

2.4.2 Applications

For this evaluation, we implement three MapReduce applications with varying characteristics, particularly the expansion factor α.

Word Count ($\alpha = 0.09$): This application takes a set of documents and computes

[3] Improving replication in geographically distributed environments would be useful for improving fault tolerance, but is outside the scope of this thesis.

the number of occurrences of each term. As input, we use plain-text eBooks from Project Gutenberg [31], totaling roughly 16.5 GB, spread across 48,000 eBooks.[4]

Sessionization ($\alpha = 1.0$): This application takes a collection of Web server logs and computes the sequence of "sessions" for each user. It is essentially a distributed sort, so the mapper simply routes data to reducers. As input, we use a portion of the WorldCup98 trace [32] (roughly 5 GB spanning 60 million log entries).

Full Inverted Index ($\alpha = 1.88$): Modeled after the example from Lin and Dyer [33], this application takes a collection of documents and computes the complete list of documents and positions where each word occurs. This application expands the input data by adding additional information, hence the high value of α. For input we use a 4 GB forward index built from the 48,000 eBooks used in the Word Count application.

2.4.3 Experimental Results

Figure 2.10 shows the results of our comparison for the three applications. For Hadoop, since the shuffle phase is partially overlapped with the map and reduce phases, we depict only three phases in each bar in the graph: the push phase, the overlapped map/shuffle phases, and the overlapped shuffle/reduce phases. We see that across all applications, while vanilla Hadoop substantially outperforms the uniform execution plan (by 68%, 40%, and 44% for Word Count, Sessionization, and Full Inverted Index respectively), Hadoop executing our optimized plan achieves a further improvement of 36%, 41%, and 31% over vanilla Hadoop for the same applications. Further, we see how vanilla Hadoop makes myopic decisions. Hadoop reduces the push time substantially (by 92%, 91%, and 83% for Word Count, Sessionization, and Full Inverted Index, respectively) over uniform. Our optimization, however, while increasing the push time over vanilla Hadoop, achieves more significant reduction in end-to-end makespan than vanilla Hadoop by better optimizing the map, shuffle, and reduce phases. Overall, we find that Hadoop executing our offline end-to-end, multi-phase optimal execution plan outperforms vanilla Hadoop using its dynamic mechanisms.

[4] At the time of writing, Project Gutenberg hosted fewer than 48,000 books. To reach this data size, we gathered 24,000 books and included each in our inputs twice.

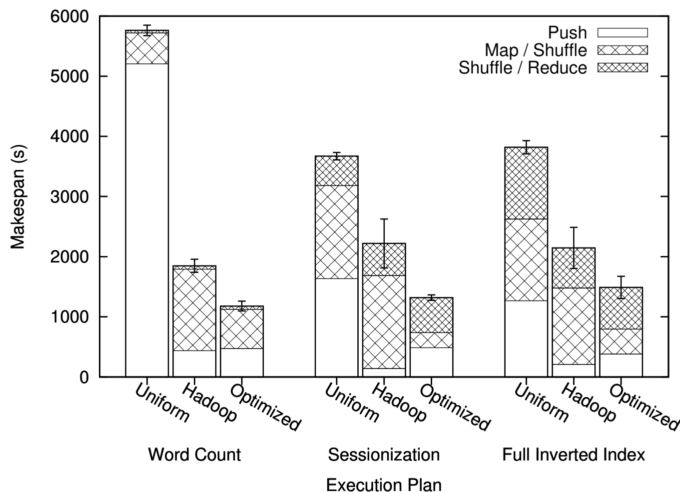


Figure 2.10: Actual makespan for three sample applications on our local testbed. Error bars reflect 95% confidence intervals.

Dynamically Enhancing the Optimized Plan

Our optimization provides an offline execution plan based on information about the underlying infrastructure available before the job begins execution. As mentioned earlier, Hadoop provides two dynamic mechanisms to modify the initial execution plan based on the observed runtime behavior of the network and nodes: (i) *speculative task execution*, where another copy of a straggler task is launched on a different node; and (ii) *work stealing*, where a node may request a non-local task if that node is idle. Here, we evaluate the benefit of using dynamic mechanisms in addition to the static execution plan derived from the results of our optimization.

Figure 2.11 shows the impact of enabling these two dynamic mechanism on the performance of the optimized plan for each of the three sample applications. Additionally, Figure 2.12 shows the impact of enabling these mechanisms atop a competitive Hadoop baseline plan, specifically one that pushes from each data source to the most local mapper and distributes intermediate keys uniformly across all reducers.

We find from these two figures that, applied on its own, speculation does not statistically significantly degrade performance in any case, while it significantly improves performance in one case. On the other hand, the addition of work stealing to speculation never statistically significantly improves performance over speculation alone, while

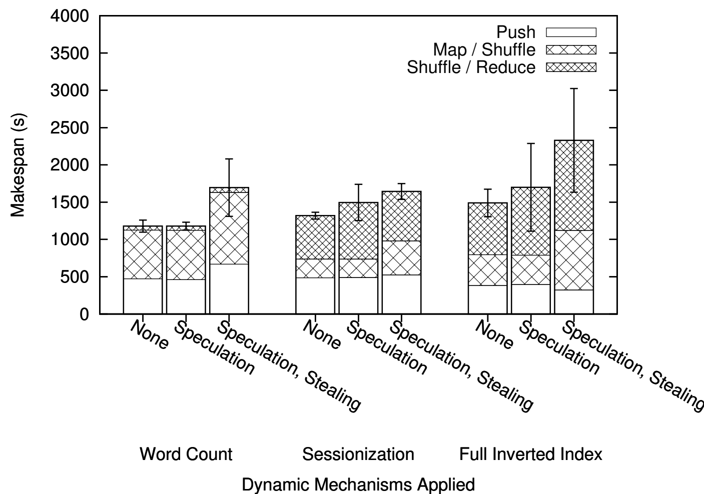


Figure 2.11: Effect of Hadoop's dynamic scheduling mechanisms applied atop our optimized static execution plan for three sample applications. Error bars reflect 95% confidence intervals.

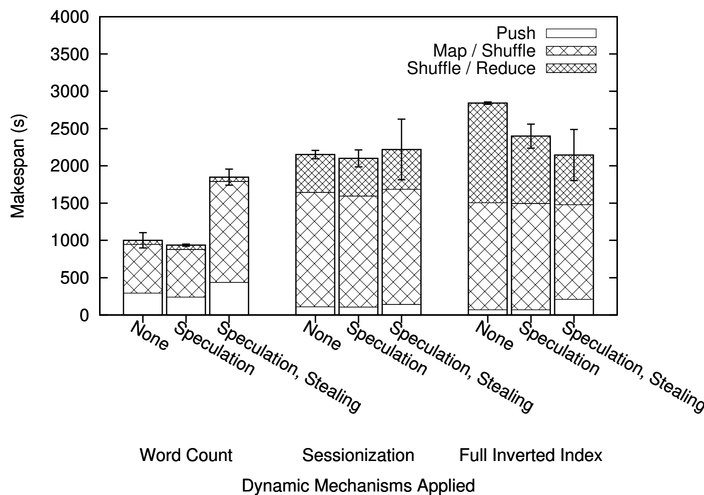


Figure 2.12: Effect of Hadoop's dynamic scheduling mechanisms applied atop a competitive Hadoop baseline plan. Error bars reflect 95% confidence intervals.

in some cases—specifically Word Count—stealing significantly degrades performance. In fact, there is only one case where the combination of speculation and stealing is statistically significantly better than the static baseline: Full Inverted Index with the Hadoop baseline. The reason in this case is that the static plan myopically optimizes the push phase, adversely impacting the much more dominant shuffle and reduce phases. By enabling speculation and optionally stealing, however, Hadoop is able to bypass the bottleneck network links and compute nodes by moving data over faster links and placing tasks on faster nodes.

Although the dynamic mechanisms are helpful in such a case, they can also degrade performance in other cases. For example, the combination of speculation and stealing statistically significantly worsens performance for two of the three applications when applied atop an optimized plan, and for one of the three applications when applied atop a Hadoop baseline plan. For the optimized plans, this occurs because dynamic changes to the offline plan can actually undermine the optimization. After all, if the computed plan is optimal, then barring any changes to the underlying infrastructure, no dynamic change could improve performance. For the Hadoop baseline, the degradation occurs for the Word Count application, for which the runtime is dominated by the push and map phases. For such an application, Hadoop's myopic optimization is actually quite effective, and dynamically deviating from this plan can yield significantly worse performance as we see here.

Though it is not shown directly in the figures, it is noteworthy that the best Hadoop performance is never statistically significantly better than the performance with the optimized plan. Hadoop's best performance relative to the optimized plan occurs for the Word Count application, for which Hadoop with speculation (but not stealing) yields a lower mean makespan than the optimized plan, but not statistically significantly so.

These results overall show the strength of the statically enforced optimized plan. At the same time, they show how dynamic mechanisms can improve performance when the initial plan is far from optimal, as is the case for the Full Inverted Index application with the Hadoop baseline. Such a situation could also arise if network or node conditions change significantly, for example due to network congestion or changes in background CPU load. Developing dynamic mechanisms to improve performance in such cases without degrading performance in other cases is an interesting direction for future work.

Impact of Data Replication Over Slow Links

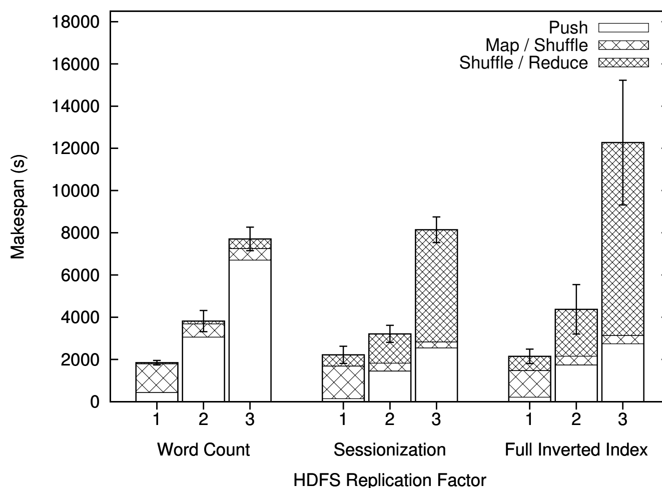


Figure 2.13: Effect of HDFS file replication for three sample applications. Error bars reflect 95% confidence intervals.

In our model, we restrict replication to be intra-cluster, to avoid sending redundant data across slow wide-area links. Figure 2.13 shows the impact of wide-area replication on the performance of vanilla Hadoop for each of the three applications. As the figure shows, increasing replication substantially increases the cost of data push, as well as the cost of the reduce phase due to the need to materialize final results to the distributed file system. While higher replication also yields a reduction in compute time in the map phase due to greater scheduling flexibility, this improvement is dwarfed by the increased communication costs. However, replication across clusters would be useful for achieving higher fault tolerance against geographically localized faults. Such replication may also provide other opportunities for performance optimization; e.g., work stealing may be directed to local tasks to avoid high-overhead data communication. Enhancing our model to incorporate cross-cluster replication, or complementing our execution with other techniques such as checkpointing for intermediate data [34] is an interesting direction for future work.

2.5 Related Work

Several other works consider data and task placement in MapReduce. Purlieus [35] categorizes jobs by the relative size of inputs to their map and reduce phases and applies different optimization approaches for each category; this is similar to our use of α as a key application parameter. CoGRS [36] considers partition skew and locality and places reduce tasks as close to their intermediate data as possible. Unlike our approach, however, it does not consider reduce task locality while placing map tasks.

Gadre et al. [37] and Kim et al. [38] focus on geographically distributed data and compute resources, respectively. Kim et al. focus on reaching the end of the shuffle phase as early as possible, whereas our objective is minimizing *end-to-end* makespan. Both of these works consider scheduling within only one phase, while our multi-phase approach controls both the map and reduce phases.

Other work considers geo-distributed MapReduce deployments from an architectural standpoint. Cardosa et al. [39] explore three architectures ranging from highly centralized to geographically distributed and conclude that no one architecture is appropriate for all applications. Luo et al. [40] propose running multiple local MapReduce jobs and aggregating their results using a new "Global Reduce" phase. They assume that communication is a small part of the total runtime, whereas our approach can find the best execution plan for a broad range of application and system characteristics.

Heterogeneity in tightly coupled local clusters has been addressed by several other works. For example Zaharia et al. [41] propose the LATE scheduler to better detect straggler tasks due to hardware heterogeneity. Mantri [42] and Tarazu [43] take *proactive* approaches to recognizing stragglers and dynamically balancing workloads, respectively. While these works focus on tightly coupled local clusters, they may also apply at the level of a single data center in the geo-distributed deployments that we consider.

Our work does not address the problem of resource provisioning or configuration, instead assuming that the set of available compute resources is determined in advance. Elastisizer [26], STEAMEngine [44], and Bazaar [27] each address the provisioning problem using a different mix of online and offline profiling, as well as predictive modeling. Starfish [45] adapts to system workloads and application characteristics to tune Hadoop configuration options.

Sandholm et al. [46] focus on multi-job scheduling in MapReduce clusters, as do Delay Scheduling [47] and Quincy [48]. While we have focused in this chapter on a single job running in isolation, extending our work to multiple concurrent jobs in a geographically distributed setting is an interesting area for future work.

Our model-driven optimization requires knowledge of network bandwidth and compute speeds. The Network Weather Service (NWS) [49] and OPEN [50] demonstrate systems for gathering and disseminating such information in a scalable manner. He et al. [51] investigate formula-based and history-based techniques for predicting the throughput of large TCP transfers, and show that history-based approaches with even a small number of samples can lead to accurate prediction.

2.6 Concluding Remarks

In this chapter, we focused on optimizing the makespan of data-intensive applications, particularly those that fit the MapReduce model, in geographically distributed environments comprising distributed data and distributed computation resources. We developed a modeling framework to capture the execution time of MapReduce applications in these environments, and showed that it is flexible enough to capture several design choices. Additionally, we developed a model-driven optimization with two key features: (i) it optimizes end-to-end makespan unlike myopic optimizations which make locally optimal but globally suboptimal decisions, and (ii) it controls multiple MapReduce phases to minimize makespan, unlike single-phase optimizations which control only individual phases. We used our model to show how our end-to-end multi-phase optimization can significantly reduce execution time compared to myopic or single-phase baselines. We also modified the Hadoop MapReduce framework to enact our optimization outputs, and used three different applications over an 8-node emulated PlanetLab testbed to show that our optimized execution plans can achieve 31–41% reduction in runtime over a vanilla Hadoop execution. Our model-driven optimization provided insights into the problem of placing data and computation in geographically distributed data intensive computing. For example, we showed that our optimization can significantly outperform either a fully centralized or fully distributed approach over a broad range of application and network characteristics.

Chapter 3

Cross-Phase Optimization in MapReduce

3.1 Introduction

The model-driven optimization from Chapter 2 leads us to the following high-level insights for geographically distributed MapReduce applications:

- Neither a purely distributed nor a purely centralized execution is the best for all situations. Instead, the best approach typically lies between either of these extremes.
- Optimizing with an end-to-end *objective* yields significantly lower makespan than optimizing with a myopic objective. The reason is that end-to-end optimization tolerates local suboptimality in order to achieve global optimality.
- Jointly *controlling* multiple MapReduce phases lowers the makespan compared to controlling only a single phase. Although optimizing the bottleneck phase alone can be beneficial, each phase contributes to the overall makespan, so controlling multiple phases has the best overall benefit, independent of application characteristics.

These high-level insights inspire the design of *cross-phase* optimization techniques that are suitable for real-world MapReduce implementations, where decisions must be

made under imperfect knowledge, and the dynamic nature of network, compute, and application characteristics might render a static optimization approach impractical.

Based on the need for an end-to-end optimization objective, our *Map-Aware Push* technique makes push decisions aware of their impact not just on the push phase, but also on the map phase. In the spirit of multi-phase optimization, we also control communication in the shuffle phase through our *Shuffle-Aware Map* technique, which factors in predicted shuffle costs while placing map tasks. This technique also contributes to a more nearly end-to-end optimization, as it allows both push and map placement to be influenced by downstream costs in the shuffle phase.

Chapter Outline

The remainder of this chapter is organized as follows. We present our proposed *Map-Aware Push* and *Shuffle-Aware Map* techniques in Sections 3.2 and 3.3, respectively. We show experimental results by putting these techniques together in an end-to-end manner in Section 3.4. We present related work in Section 3.5 and conclude in Section 3.6.

3.2 Map-Aware Push

The first opportunity for cross-phase optimization in MapReduce lies at the boundary between the push and map phases. A typical practice is what we call a *push-then-map* approach, where input data are imported in one step, and computation begins only after the data import completes. This approach has two major problems. First, by forcing all computation to wait for the slowest communication link, it introduces waste due to idle resources. Second, separating the push and map phases deprives map tasks of a way to demand more or less work based on their compute capacity. This makes scheduling the push in a map-aware manner more difficult. To overcome these challenges, we propose two changes: first, overlapping—or pipelining—the push and map phases; and second, inferring locality information at runtime and driving scheduling decisions based on this knowledge.

Before discussing how we implement our proposed approach in Hadoop and showing experimental results, we describe in more detail the problems of a push-then-map approach and how our proposed *Map-Aware Push* technique addresses them. To begin,

consider the simple example environment shown in Figure 3.1, comprising two data sources S_1 and S_2 and two mappers M_1 and M_2. Assume that each data source initially hosts 15 GB of data and that the link bandwidths and mapper computation rates are as shown in the figure.

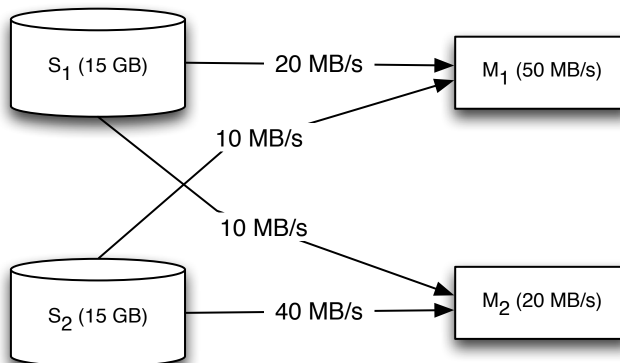


Figure 3.1: A simple example network with two data sources and two mappers.

3.2.1 Overlapping Push and Map to Hide Latency

With these network links, the following push distribution would optimize the push (i.e., minimize push time): S_1 pushes 10 GB to M_1 and 5 GB to M_2, and S_2 pushes 3 GB to M_1 and 12 GB to M_2. If we were to use this distribution with a push-then-map approach, then the entire push would finish after 500 s even though source S_2 would finish its part after only 300 s. Map computation would begin after 500 s and continue until 1350 s.

If we instead were to overlap the push and map, allowing map computation to begin at each mapper as soon as data begin to arrive, then we could avoid this unnecessary waiting. For example, assuming that we used this same optimal push distribution, then mapper M_2, with slower compute capacity than its incoming network links, would be the bottleneck, finishing after 850 s. By simply overlapping the map and push phases, we could reduce the total push and map runtime by about 37% in this example.

3.2.2 Overlapping Push and Map to Improve Scheduling

The second problem arises due to the lack of feedback from the map phase to the push phase. Without this feedback, and absent any *a priori* knowledge of the map phase performance, we are left with few options other than simply optimizing the push phase in isolation. Such a single-phase optimization favors pushing more data to mappers with faster incoming network links. In our example, however, it is the mapper with *slower* network links (M_1) that is actually better suited for map computation. Unfortunately, by pursuing an optimal push phase, we end up with more data at M_2 and in turn roughly 3.3x longer map computation there than at M_1. For better overall performance, we need to weigh the two factors of network bandwidth and computation capacity and trade off between faster push and faster map. Continuing with our simple example, we should tolerate a slightly slower push in order to achieve a significantly faster map by sending more data to mapper M_1. In fact, in an optimal case, we would send 60% of all input data there, yielding a total push-map runtime of only 600 s, or a 55% reduction over the original push-then-map approach.

3.2.3 Map-Aware Push Scheduling

This raises the question of how we can schedule push and map jointly, respecting the interaction between the two phases. As we have argued, this is difficult to do when the push and map phases are separated. With overlapped push and map, however, the distribution of computation across mapper nodes can be demand-driven. Specifically, whereas push-then-map first pushes data from sources, our approach logically *pulls* data from sources on-demand. Using existing Hadoop mechanisms, this on-demand pull is initiated when a mapper becomes idle and requests more work, so faster mappers can perform more work. This is how our proposed approach respects map computation heterogeneity.

To respect network heterogeneity, our *Map-Aware Push* technique departs from the traditional Hadoop approach of explicitly modeling network topology as a set of racks and switches, and instead infers locality information at runtime. It does this by monitoring source-mapper link bandwidth at runtime and estimating the push time for each source-mapper pair. Specifically, let d be the size of a task in bytes (assume for ease of

presentation that all task sizes are equal) and let $L_{s,m}$ be the link speed between source node s and mapper node m in bytes per second. Then we estimate the push time $T_{s,m}$ in seconds from source s to mapper m as

$$T_{s,m} = \frac{d}{L_{s,m}} \ . \tag{3.1}$$

Let S denote the set of all sources that have not yet completed their push. Then when mapper node m requests work, we grant it a task from source $s^* = \text{argmin}_{s \in S} T_{s,m}$. Intuitively, this is equivalent to selecting the closest task in terms of network bandwidth. This is a similar policy to Hadoop's default approach of preferring *data-local* tasks, but our overall approach is distinguished in two ways. First, rather than *reacting* to data movement decisions that have already been made in a separate push phase, it *proactively* optimizes data movement and task placement in concert. Second, it discovers locality information dynamically and automatically rather than relying on an explicit user-specified model.

3.2.4 Implementation in Hadoop

Now we can discuss how we have implemented our approach in Hadoop 1.0.1. First, the overlapping itself is possible using existing Hadoop mechanisms, but a more creative deployment. Specifically, we set up a Hadoop Distributed File System (HDFS) instance comprising the data source nodes, which we refer to as the "remote" HDFS instance and use directly as the input to a Hadoop MapReduce job. Map tasks in Hadoop typically read their inputs from HDFS, so this allows us to directly employ existing Hadoop mechanisms.[1]

Our scheduling enhancements, on the other hand, require modification to the Hadoop task scheduler. To gather the bandwidth information mentioned earlier, we add a simple network monitoring module which records actual source-to-mapper link performance and makes this information accessible to the task scheduler. For Hadoop MapReduce jobs that read HDFS files as input, each map task corresponds to an `InputSplit` which in turn corresponds to an HDFS file block. HDFS provides an interface to determine

[1] To improve fault tolerance, we have also added an option to cache and replicate inputs at the compute nodes. This reduces the need to re-fetch remote data after task failures or for speculative execution.

physical block locations, so the task scheduler can determine the source associated with a particular task and compute its $T_{s,m}$ based on bandwidth information from the monitoring module. If there are multiple replicas of the file block, then $T_{s,m}$ can be computed for each replica, and the system can use the replica that minimizes this value. The task scheduler then assigns tasks from the closest source s^* as described earlier.

3.2.5 Experimental Results

We are interested in the performance of our approach, which overlaps push and map and infers locality at runtime, compared to a baseline push-then-map approach. To implement the push-then-map approach, we also run an HDFS instance comprising the compute nodes (call this the "local" HDFS instance). We first run a Hadoop `DistCP` job to copy from the remote HDFS to this local HDFS, and then run a MapReduce job directly from the local HDFS. We compare application execution time using these two approaches. Because we are concerned primarily with push and map performance at this point, we run the Hadoop example WordCount job on text data generated by the Hadoop example `randomtextwriter` generator, as this represents a map-heavy application.

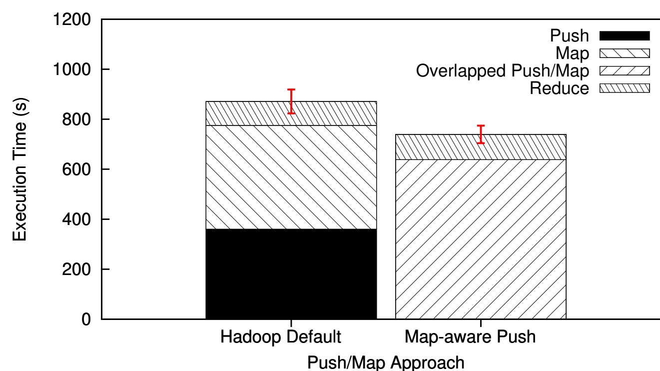
We run this experiment in two different environments: Amazon EC2 and PlanetLab. Our EC2 setup uses eight EC2 nodes in total, all of the `m1.small` instance type. These nodes are distributed evenly across two EC2 regions: four in the US and the other four in Europe. Each node hosts one map slot and one reduce slot. Two PlanetLab nodes, one in the US and one in Europe, serve as distributed data sources. Table 3.1 shows the bandwidths measured between the multiple nodes in this setup.

Figure 3.2(a) shows the execution time of the WordCount job on 2 GB of input data, and it shows that our approach to overlapping push and map reduces the total runtime of the push and map phases by 17.7%, and the total end-to-end runtime by 15.2% on our EC2 testbed.

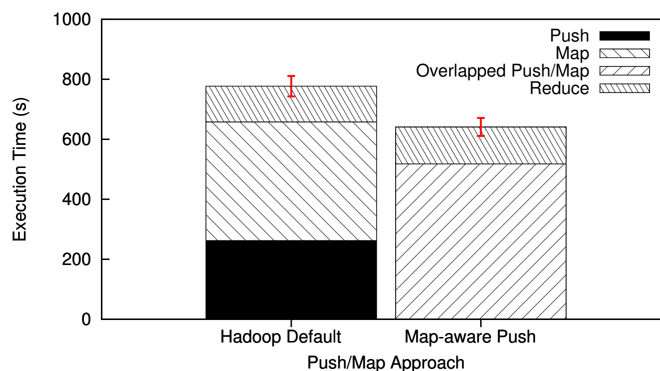
Next, we run the same experiment on PlanetLab. We continue to use two nodes as distributed data sources, and we use four other globally distributed nodes as compute nodes, each hosting one map slot and one reduce slot. Table 3.2 shows the bandwidths measured between the multiple nodes in this setup. Due to the smaller cluster size in this experiment, we use only 1 GB of text input data.

Table 3.1: Measured bandwidths in the EC2 experimental setup.

From	To	Bandwidth (MB/s)
Source EU	Worker EU	8
Source EU	Worker US	3
Source US	Worker EU	3
Source US	Worker US	4
Worker EU	Worker EU	16
Worker EU	Worker US	2
Worker US	Worker EU	5
Worker US	Worker US	2



(a) WordCount on 2 GB text data on EC2



(b) WordCount on 1 GB text data on PlanetLab

Figure 3.2: Runtime of a Hadoop WordCount job on text data for the push-then-map approach and the *Map-Aware Push* approach on globally distributed Amazon EC2 and PlanetLab test environments. Error bars reflect 95% confidence intervals.

Table 3.2: Measured bandwidths in the PlanetLab experimental setup.

From	To	Bandwidth (MB/s)
All sources	All workers	1-3
Workers A-C	Workers A-C	4-9
Workers A-C	Worker D	2
Worker D	Workers A-C	0.2-0.4

Figure 3.2(b) shows that push-map overlap can reduce runtime of the push and map phases by 21.3% and the whole job by 17.5% in this environment. We see a slightly greater benefit from push-map overlap on PlanetLab than on EC2 due to the increased heterogeneity of the PlanetLab environment.

3.3 Shuffle-Aware Map

In the previous section, we showed how the map phase can influence the push phase, in terms of both the volume of data each mapper receives as well as the sources from which each mapper receives its data. In turn, the push determines, in part, when a map slot becomes available for a mapper. Thus, from the perspective of the push and map phases, a set of mappers and their data sources are decided. This decision, however, ignores the downstream cost of the shuffle and reduce as we will show. In this section, we show how the set of mappers can be adjusted to account for the downstream shuffle cost.

In traditional MapReduce, intermediate map outputs are shuffled to reducers in an all-to-all communication. In Hadoop, one can control the granularity of reduce tasks and the amount of work each reducer will obtain. However, these decisions ignore the possibility that a mapper-reducer link may be very poor. For example, in Figure 3.3, the links between mapper C and reducers D and E are poor, thus raising the cost of shuffle. For applications in which shuffle is dominant, this phenomenon can greatly impact performance, particularly in heterogeneous networks.

Two solutions are possible: changing the reducer nodes, or reducing the amount of work done by mapper C and in turn reducing the volume of data traversing the bottleneck links. We present an algorithm that takes the latter approach. In this way,

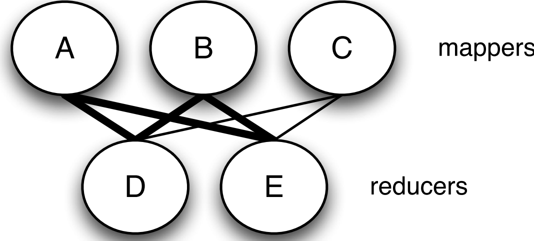


Figure 3.3: An example network where links from mapper C to reducers D and E are shuffle bottlenecks.

the downstream shuffle (or reduce) can impact the map. This is similar to the *Map-Aware Push* technique where the map influenced the push.

As in typical MapReduce, we assume the reducer nodes are known *a priori*. We also assume that we know the approximate distribution of reducer tasks: i.e., we know the fraction of intermediate data allocated to each reducer node. This allows us to know how much data must travel on the link from a mapper node to each reducer, which our algorithm utilizes. The distribution can be estimated using a combination of reducer node execution power and mapper-reduce link speed, pre-profiled. This estimate can be updated during the map phase if shuffle and reduce are overlapped.

3.3.1 Shuffle-Aware Map Scheduling

To estimate the impact of a mapper node upon the reduce phase, we first estimate the time taken by the mapper to obtain a task, execute it, and deliver intermediate data to all reducers (assuming parallel transport). The intuition is that if the shuffle cost is high, then the mapper node should be throttled to allow the map task to be allocated to a mapper with better shuffle performance. We estimate the finish time T_m for a mapper m to execute a map task as follows: $T_m = T_m^{\text{map}} + T_m^{\text{shuffle}}$, where T_m^{map} is the estimated time for the mapper m to execute the map task, including the time to read the task input from a source (using the *Map-Aware Push* approach), and T_m^{shuffle} is the estimated time to shuffle the accumulated intermediate data D_m up to the current task, from mapper m to all reducer nodes $r \in R$, where R is the set of all reducer nodes. Let $D_{m,r}$ be the portion of D_m destined for reducer r, and $L_{m,r}$ be the link speed between

mapper node m and reducer node r. Then, we can compute

$$T_m^{\text{shuffle}} = \max_{r \in R} \left(\frac{D_{m,r}}{L_{m,r}} \right). \tag{3.2}$$

The *Shuffle-Aware Map* scheduling algorithm uses these T_m estimates to determine a set of *eligible mappers* to which to assign tasks. The intuition is to throttle those mappers that would have an adverse impact on the performance of the downstream reducers. The set of eligible mappers M_{eligible} is based on the most recent T_m values and a tolerance parameter α:

$$M_{\text{eligible}} = \{m \in M | T_m \leq \min_{m \in M} T_m + \alpha\}, \tag{3.3}$$

where M is the set of all mapper nodes.

The intuition is that if the execution time for a mapper (including its shuffle time) is too high, then it should not be assigned more work at present. The value of the tolerance parameter α controls the aggressiveness of the algorithm in excluding slower mappers (in terms of their shuffle performance) from being assigned work. At one extreme, $\alpha = 0$ would enforce assigning work only to the mapper with the earliest estimated finish time, intuitively achieving good load balancing, but leaving all other mappers idle for long periods of time. At the other extreme, a high value of $\alpha \geq (\max_{m \in M} T_m - \min_{m \in M} T_m)$ would allow all mapper nodes to be eligible irrespective of their shuffle performance, and would thus reduce to the default MapReduce map scheduling. We select an intermediate value:

$$\alpha = \frac{(\max_{m \in M} T_m - \min_{m \in M} T_m)}{2}. \tag{3.4}$$

The intuition behind this value is that it biases towards roughly the upper half of mappers in terms of their shuffle performance. This is of course just one possible threshold; exploring other possibilities would be an interesting topic for future work.

One point worth emphasizing is that the algorithm makes its decisions dynamically, so that over time, a mapper may become eligible or ineligible depending on the relation between its T_m value and the current value of $\min_{m \in M} T_m$. As a result, this algorithm allows a discarded mapper node to be re-included later should other nodes begin to offer worse performance. Similarly, a mapper may be throttled if its performance starts degrading over time.

3.3.2 Implementation in Hadoop

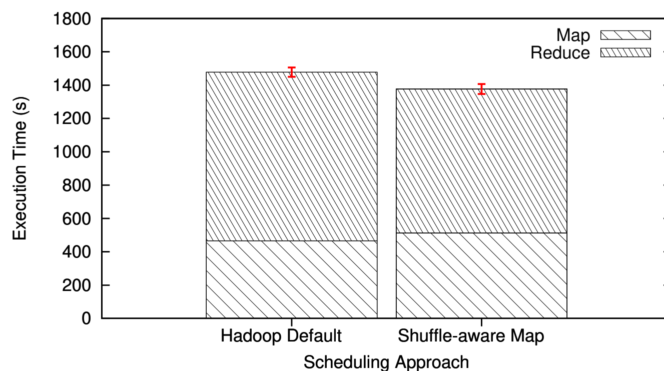
We have implemented this *Shuffle-Aware Map* scheduling algorithm by modifying the task scheduler in Hadoop. The task scheduler now maintains a list of estimates T_m for all mapper nodes m, and updates these estimates as map tasks finish. It also uses the mapper-to-reducer node pair bandwidth information obtained by the network monitoring module to update the estimates of shuffle times from each mapper node. Every time a map task finishes, the task tracker on that node asks the task scheduler for a new map task. At that point, the scheduler applies Equation 3.3 to determine the eligibility of the node to receive a new task. If the node is eligible, then it is assigned a task from the best source determined by the *Map-Aware Push* algorithm described in Section 3.2. On the other hand, if the node is not eligible, then it is not assigned a task. However, it can request work again periodically by piggybacking on heartbeat messages, and its eligibility will be checked again.

3.3.3 Experimental Results

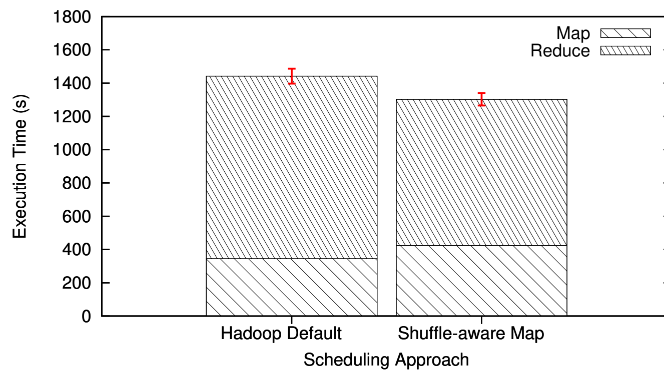
We now present some results that show the benefit of *Shuffle-Aware Map*. Here we run our InvertedIndex application, which takes as input a set of eBooks from Project Gutenberg [31] and produces, for each word in its input, the complete list of positions where that word can be found. This application shuffles a large volume of intermediate data, so it is an interesting application for evaluating our *Shuffle-Aware Map* scheduling technique.

First, we run this application on our EC2 multi-region cloud as described in Table 3.1. In this environment, we use 1.8 GB of eBook data as input, and this produces about 4 GB of intermediate data to be shuffled to reducers. Figure 3.4(a) shows the runtime for a Hadoop baseline with push and map overlapped, as well as the runtime of our *Shuffle-Aware Map* scheduling technique, also with push and map overlapped.

The reduce time shown includes shuffle cost. Note that in *Shuffle-Aware Map* the shuffle and reduce time (labeled “reduce” in the figure) are smaller than in default Hadoop. Also observe that in *Shuffle-Aware Map* the map times go up slightly—this algorithm has decided to make this tradeoff resulting in overall better performance. On our wider-area PlanetLab setup (see Table 3.2) we use 800 MB of eBook data and see



(a) InvertedIndex on 1.8 GB eBook data on EC2



(b) InvertedIndex on 800 MB eBook data on PlanetLab

Figure 3.4: Runtime of the InvertedIndex job on eBook data for the default Hadoop scheduler and our *Shuffle-Aware Map* scheduler. Both approaches use an overlapped push and map in these experiments. Error bars reflect 95% confidence intervals.

a similar pattern, as Figure 3.4(b) shows. Again, an increase in map time is tolerated to reduce shuffle cost later on. This may mean that a slower mapper is given more work since it has faster links to downstream reducers. For this application, we see performance improvements of 6.8% and 9.6% on EC2 and PlanetLab, respectively.

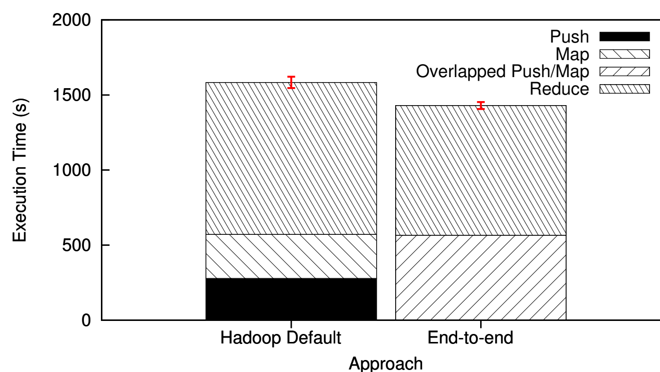
3.4 Putting it all Together

To determine the complete end-to-end benefit of our proposed techniques, we run experiments comparing a traditional Hadoop baseline, which uses a push-then-map approach, to an alternative that uses our proposed *Map-Aware Push* and *Shuffle-Aware Map* techniques. Taken together, we will refer to our techniques as the *End-to-end* approach. We carry out these experiments on the same PlanetLab and EC2 test environments introduced in Table 3.1 and Table 3.2, respectively. We focus here on the InvertedIndex application from Section 3.3 as well as a new Sessionization application. This Sessionization application takes as input a set of Web server logs from the WorldCup98 trace [32], and sorts these records first by client and then by time. The sorted records for each client are then grouped into a set of "sessions" based on the gap between subsequent records. Both the InvertedIndex and Sessionization applications are relatively shuffle-heavy, representing a class of applications that can benefit from our *Shuffle-Aware Map* technique.

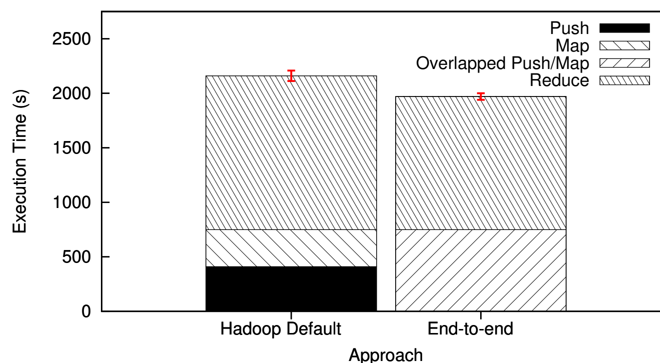
3.4.1 Amazon EC2

First, we explore the combined benefit of our techniques on our EC2 test environment (see Section 3.2 for details), comprising two distributed data sources and eight worker nodes spanning two EC2 regions. Figure 3.5(a) shows results for the InvertedIndex application, where we see that our approaches reduce the total execution time by about 9.7% over the traditional Hadoop approach. There is little difference in total push and map time, so most of this reduction in runtime comes from a faster shuffle and reduce (labeled "reduce" in the figure). This demonstrates the effectiveness of our *Shuffle-Aware Map* scheduling approach, as well as the ability of our techniques to automatically determine how to tradeoff between faster push and map phases or faster shuffle and reduce phases.

Now consider the Sessionization application, which has a slightly lighter shuffle and slightly heavier reduce than does the InvertedIndex application. Figure 3.5(b) shows that for this application on our EC2 environment, our approaches can reduce execution time by 8.8%. Again most of the reduction in execution time comes from more efficient shuffle and reduce phases. Because this application has a slightly lighter shuffle than does the InvertedIndex application, we would expect a slightly smaller performance improvement, and our experiments confirm this.



(a) InvertedIndex on 1.8 GB eBook data on EC2



(b) Sessionization on 2 GB text log data on EC2

Figure 3.5: Execution time for traditional Hadoop compared with our proposed *Map-Aware Push* and *Shuffle-Aware Map* techniques (together, *End-to-end*) for InvertedIndex and Sessionization applications on our EC2 test environment. Error bars reflect 95% confidence intervals.

3.4.2 PlanetLab

Now we move to the PlanetLab environment, which exhibits more extreme heterogeneity than the EC2 environment. For this environment, we consider only the InvertedIndex application, and Figure 3.6 shows that our approaches can reduce execution time by about 16.4%. Although we see a slight improvement in total push and map time using our approach, we can again attribute the majority of the performance improvement to a more efficient shuffle and reduce.

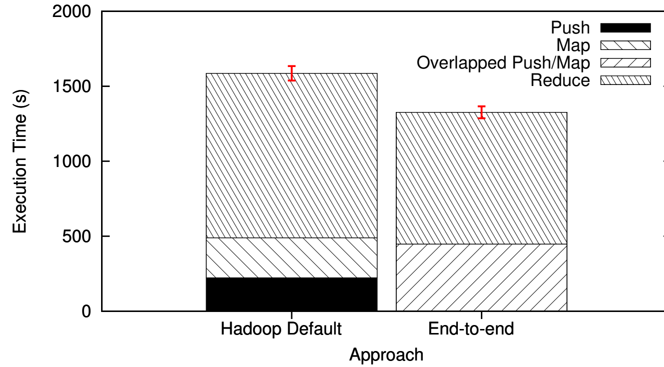


Figure 3.6: Execution time for traditional Hadoop compared with our proposed *Map-Aware Push* and *Shuffle-Aware Map* techniques (together, *End-to-end*) for for the InvertedIndex application with 800 MB eBook data on our PlanetLab test environment. Error bars reflect 95% confidence intervals.

Table 3.3: Number of map tasks assigned to each mapper node in our PlanetLab test environment.

Scheduler	Mapper A	Mapper B	Mapper C	Mapper D
Hadoop Default	5	4	5	3
End-to-end	5	5	6	1

To more deeply understand how our techniques achieve this improvement, we record the number of map tasks assigned to each mapper node, as shown in Table 3.3. We see that both Hadoop and our techniques assign fewer map tasks to Mapper D, but that our techniques do so in a much more pronounced manner. Network bandwidth measurements reveal that this node has much slower outgoing network links than do the other mapper nodes; only about 200–400 KB/s compared to about 4–9 MB/s for the

other nodes (see Table 3.2). By scheduling three map tasks at that node, Hadoop has effectively "trapped" intermediate data there, resulting in a prolonged shuffle phase. Our *Shuffle-Aware Map* technique, on the other hand, has the foresight to avoid this problem by refusing to grant Mapper D additional tasks even when it becomes idle and requests more work.

3.5 Related Work

Heterogeneity

Traditionally, the MapReduce [13] programming paradigm assumes a tightly coupled homogeneous cluster applied to a uniform data-intensive application. Previous work has shown that if this assumption is relaxed, then performance suffers. Zaharia et al. [41] show that, under computational heterogeneity, the mechanisms built into Hadoop for identifying straggler tasks break down. Their LATE scheduler provides better techniques for identifying, prioritizing, and scheduling backup copies of slow tasks. In our work, we also assume that nodes can be heterogeneous since they belong to different data centers or locales. Chen et al. [52] report techniques for improving the accuracy of progress estimation for tasks in MapReduce. Ahmad et al. [43] demonstrate that despite straggler optimizations, the performance of MapReduce frameworks on clusters with computational heterogeneity remains poor as the load balancing used in MapReduce causes excessive and bursty network communication and the heterogeneity further amplifies the load imbalance of reducers. Their Tarazu system uses a communication-aware balancing mechanism and predictive load-balancing across reducers to address these problems. Mantri [42] explores various causes of outlier tasks in further depth, and develops cause- and resource-aware techniques to identify and act on outliers earlier, and to greater benefit, than in traditional MapReduce. Such improvements are complementary to our techniques. Previous work mainly focuses on computational heterogeneity within a single cluster. Our work, however, targets more loosely coupled and dispersed collections of resources with constrained and heterogeneous bandwidth.

Loosely-Coupled Deployments

Several works have targeted MapReduce deployments in loosely coupled environments. MOON [53] explores MapReduce performance in volatile, volunteer computing environments and extends Hadoop to improve performance under loosely coupled networks with unreliable slave nodes. In our work, we do not focus on reliability; instead we are concerned with performance issues of allocating compute resources to MapReduce tasks. Costa et al. [54] propose MapReduce in a global wide-area volunteer setting. However, this system is implemented in the BOINC framework with all input data held by the central scheduler. In our system, we have no such restrictions. Luo et al. [40] propose a multi-cluster MapReduce deployment, but they focus on more compute-intensive jobs that may require resources in multiple clusters for greater compute power. In contrast, we consider multi-site resources not only for their compute power, but also for their locality to data sources.

Data Flow and Locality

Other researchers have addressed MapReduce data flow optimization and locality. Gadre et al. [37] optimize the reduce data placement according to map output locations, which might still end up trapping data in nodes with slow outward links. Kim et al. [38] present a similar idea of shuffle-aware scheduling, but do not consider widely distributed data sources that are not co-located with computation clusters. Their ICMR algorithm could make use of our mechanisms to improve the performance of MapReduce under such environments. Pipelining MapReduce has been proposed in MapReduce Online [28] to modify the Hadoop workflow for improved responsiveness and performance. It assumes, however, that input data are located with the computation resources, and it does not address the issue of pipelining push and map. MapReduce Online would be a complementary optimization to our techniques since it enables the shuffling of intermediate data without storing it to disk. Our mechanisms could be used to decide where data should flow, while their techniques could be used to optimize the transfer. Similarly, Verma et al. [29] discuss the specific challenges associated with pipelining the shuffle and reduce stages. Our *Map-aware Push* technique could also be applied to pipeline shuffle and reduce, though we have not yet done so. The Purlieus system [35] considers

MapReduce in a single cloud, but is unique in that it focuses on locality in the shuffle phase. It emphasizes the coupling between the placement of tasks (in their case virtual machines) and data. However, these works do not provide an end-to-end overall improvement of the MapReduce data flow.

Parameter Tuning

Other work has focused on fine-tuning MapReduce parameters or offering scheduling optimizations to provide better performance. Sandholm et al. [46] present a dynamic prioritization system for improved MapReduce runtime in the context of multiple jobs. Our work is concerned with optimizing a single job relative to data source and compute resource locations. Babu [55] proposes algorithms for automatically fine-tuning MapReduce parameters to optimize job performance. Starfish [45] proposes a self-tuning architecture which monitors runtime performance of Hadoop and tunes the configuration parameters accordingly. Such work is complementary to ours, however, as we focus on mechanisms to directly change task and data placement rather than tune configuration parameters.

Data Transfer Protocols

Finally, work in wide-area data transfer and dissemination includes GridFTP [56] and BitTorrent [57]. GridFTP is a protocol for high-performance data transfer over high-bandwidth wide-area networks, and BitTorrent is a peer-to-peer file sharing protocol for wide-area distributed systems. These protocols could act as middleware services to further reduce data transfer costs and make geo-distributed data more accessible to geo-distributed compute resources.

3.6 Concluding Remarks

In this chapter, we showed that in geo-distributed environments, MapReduce/Hadoop performance suffers, as the impact of one phase upon another can severely degrade performance along bottleneck links. We showed that Hadoop's default data placement and scheduling mechanisms do not take such cross-phase interactions into account, and while they may try to optimize individual phases, they can result in globally bad decisions,

resulting in poor overall performance. To overcome these limitations, we developed techniques to achieve cross-phase optimization in MapReduce. The key idea behind our techniques is to consider not only the execution cost of an individual task or computational phase, but also its impact on the performance of subsequent phases. The *Map-Aware Push* technique enables push-map overlap in order to hide latency and enable dynamic feedback between the map and push phases. Such feedback enables nodes with higher speeds and faster links to process more data at runtime. The *Shuffle-Aware Map* technique enables a shuffle-aware scheduler to feed back the cost of a downstream shuffle into the map process and affect the map phase. Mappers with poor outgoing links to reducers are throttled, eliminating the impact of mapper-reducer bottleneck links. For a range of heterogeneous environments (multi-region Amazon EC2 and PlanetLab) and diverse data-intensive applications (WordCount, InvertedIndex, and Sessionization) we demonstrated the performance potential of our techniques, as they reduce execution time by 7%–18% depending on the execution environment and application.

Chapter 4

Optimizing Timeliness and Cost in Geo-Distributed Streaming Analytics

4.1 Introduction

A modern analytics service must provide near-real-time analysis of rapid and unbounded streams of data in order to extract meaningful and timely information for its users. As a result, there has been a growing interest in streaming analytics, including the recent development of several distributed analytics platforms [58, 59, 60].

In many streaming analytics domains, inputs originate from diverse sources including users, devices, and sensors located around the globe. As a result, the distributed infrastructure of a typical analytics service (e.g., Google Analytics, Akamai Media Analytics, etc.) has a hub-and-spoke model as shown in Figure 4.1. Data sources send streams of data to nearby "edge" servers. These geographically distributed edge servers process incoming data and send summaries to a central location that can process the data further, store summaries, and present those summaries in visual form to users of the analytics service. While the central hub is typically located in a well-provisioned data center, resources may be limited at the edge locations. In particular, the available WAN bandwidth between the edge and the center is limited.

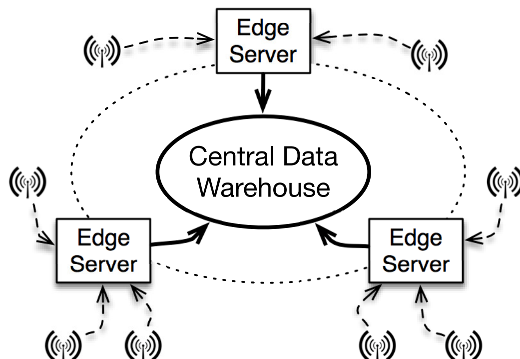


Figure 4.1: The distributed model for a typical analytics service comprises a single center and multiple edges, connected by a wide-area network.

A traditional approach to analytics processing is the *centralized model* where no processing is performed at the edges and all the data is sent to a dedicated centralized location. This approach is generally suboptimal, because it strains the scarce WAN bandwidth available between the edge and the center, leading to delayed results. Further, it fails to make use of the available compute and storage resources at the edge. An alternative is a *decentralized approach* [12] that utilizes the edge for much of the processing in order to minimize WAN traffic. In this chapter, we argue that analytics processing must utilize *both* edge and central resources in a carefully coordinated manner in order to achieve the stringent requirements of an analytics service in terms of both network traffic and user-perceived delay.

An important primitive in any analytics system is *grouped aggregation*. Abstractions for grouped aggregation are provided in most data analytics frameworks, for example as the `Reduce` operation in MapReduce, or `Group By` in SQL and LINQ. A useful variant in stream computing is *windowed* grouped aggregation, where each group is further broken down into finite time windows before being summarized. Windowed grouped aggregation is one of the most frequently used primitives in an analytics service and underlies queries that aggregate a metric of interest over a time window. For instance, a web analytics user may wish to compute the total visits to her web site broken down by country and aggregated on an hourly basis to track content popularity. A network operator may need to compute the average load in different parts of the network every five minutes to identify hotspots. In these cases, users define a standing windowed

grouped aggregation query that generates results periodically for each time window (every hour, five minutes, etc.).

This chapter focuses on designing algorithms for performing windowed grouped aggregation in order to optimize the two key metrics of any geo-distributed streaming analytics service: *WAN traffic* and *staleness* (the delay in receiving the result for a time window). While much of the existing work on decentralized analytics [8, 12] has focused primarily on optimizing a single metric (e.g., network traffic), it is important to examine both traffic and staleness together to achieve both cost savings and improved timeliness. The key decision that our algorithms make is *how much of the data aggregation should be performed at the edge versus the center.*

4.1.1 Challenges in Optimizing Windowed Grouped Aggregation

To understand the challenge of geo-distributed windowed grouped aggregation, consider two alternate approaches to grouped aggregation: *pure streaming*, where all data is immediately sent from the edge to the center without any edge processing; and *pure batching*, where all data during a time window is aggregated at the edge, with only the aggregated results being sent to the center at the end of the window. To illustrate the complexities, we consider two simplified queries from our web analytics example. For each example below, assume that the edge can communicate at the rate of 1,000 records/second to the center.

Example 4.1. Consider a query that uses grouped aggregation with key-value pairs ($\text{key} = \text{country}, \text{value} = \text{visits}$) for a time window of one hour. We say this query is "small" since the number of possible distinct keys is small as there are no more than 200 countries in the world. For this query, the right approach might be a pure batching approach. This approach will minimize the WAN traffic, since at most one record is sent for each distinct key. Further, the staleness of the final result might also be small since there are at most 200 records to send, so the aggregate values can be sent to the center within $200/1000 = 0.2$ seconds of the end of the window.

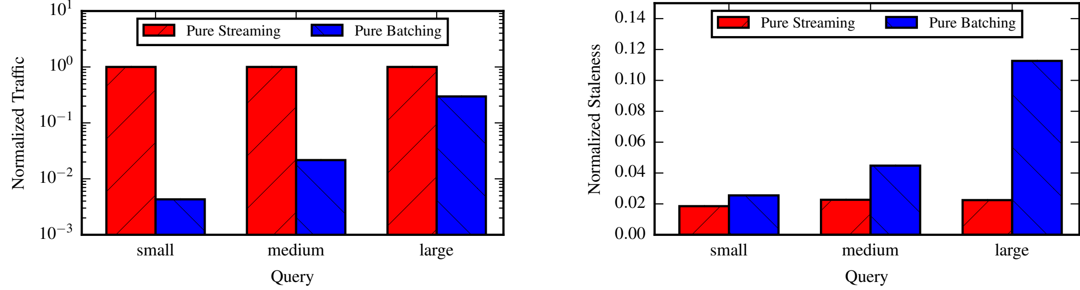
Example 4.2. Consider a second query that uses grouped aggregation with key-value pairs ($\text{key} = \langle\text{userid}, \text{url}\rangle, \text{value} = \text{visits}$) for a time window of one hour. We say that this query is "large" since the product of the unique users and urls for a large website,

and hence, the number of possible distinct keys received within a time window, could be large. Let us assume this number to be 100,000 distinct keys per window. The above batching solution may not work well for the larger query, since sending 100,000 records at the end of the time window can incur a delay of 100 seconds, resulting in higher staleness. A more appropriate approach might be pure streaming. Since each unique user/url combination is less likely to repeat within a time window, the extra traffic sent by streaming is small. Further, there is no backlog of keys that must be updated at the end of the time window, resulting in smaller staleness.

These examples illustrate that the right strategy for optimizing grouped aggregation depends on the data and query characteristics, including the rate of arrivals at the edge, the number of unique keys seen in a time window, and the available edge-to-center bandwidth. To show that the intuition gained from these examples applies in practice, we run experiments on a 5-node PlanetLab [14] setup using the traces of a popular web analytics service offered by a large CDN (see Section 4.3 for details on the dataset and queries used in our experiments). Figure 4.2 shows the traffic and staleness obtained for different queries (small, medium, and large) obtained for the pure streaming and pure batching algorithms. The figure shows that, depending on the query, pure batching can reduce traffic by 70–90% while increasing staleness by 37–300% relative to pure streaming. This illustrates that simple aggregation algorithms cannot optimize both traffic and staleness at the same time, instead optimizing one at the cost of degrading the other. Further, the factors influencing the performance of such algorithms are often hard to predict and can vary significantly over time, requiring the design of algorithms that can adapt to changing factors in a dynamic fashion.

4.1.2 Research Contributions

- To our knowledge, we provide the first algorithms and analysis for optimizing grouped aggregation, a key primitive, in a geographically distributed streaming analytics service. In particular, we show that simpler approaches such as pure streaming or batching do not jointly optimize traffic and staleness, and are hence suboptimal.



(a) Traffic normalized relative to Pure Streaming. Note the logarithmic y-axis scale.

(b) Staleness normalized by the window length.

Figure 4.2: Simple aggregation algorithms such as streaming and batching optimize one metric over the other and their performance depends on the data and query characteristics.

- We present a family of optimal offline algorithms that *jointly minimize both staleness and traffic*. Using this as a foundation, we develop practical online aggregation algorithms that emulate the offline optimal algorithms.
- We observe that grouped aggregation can be modeled as a *caching problem* where the cache size varies over time. This key insight allows us to exploit well-known caching algorithms in our design of online aggregation algorithms.
- We demonstrate the practicality of these algorithms through an implementation in Apache Storm [58], deployed on a PlanetLab [14] testbed. Our experiments are driven by workloads derived from anonymized traces of a popular web analytics service offered by Akamai [9], a large content delivery network. The results of our experiments show that our online aggregation algorithms simultaneously achieve traffic within 2.0% of optimal while reducing staleness by 65% relative to batching. We also show that our algorithms are robust to a variety of system configurations (number of edges), stream arrival rates, and query types.

4.2 Problem Formulation

System Model

We consider the typical hub-and-spoke architecture of an analytics system with a center and multiple edges (see Figure 4.1). Data streams are first sent from each source to a nearby edge. The edges collect and (potentially, partially) aggregate the data. The aggregated data can then be sent from the edges to the center for final aggregation. The final aggregated results are available at the center. Users of the analytics service query the center to visualize the data. To perform grouped aggregation, each edge runs a local aggregation algorithm: it acts independently to decide when and how much to aggregate the incoming data. (Coordination between edges is an interesting topic [61] for future work, but is outside scope of this thesis.)

Data Streams and Grouped Aggregation

A *data stream* comprises *records* of the form (k, v) where k is the *key* and v is the *value* of the record. Data records of a stream *arrive* at the edge over time. Each key k can be multi-dimensional, with each dimension corresponding to a data attribute. A *group* is a set of records that have the same key.

Windowed grouped aggregation over a *time window* $[T, T+W)$, where W is the window length, is defined as follows from an input/output perspective. The input is the set of data records that arrive within the time window. The output is determined by first placing the data records into groups where each group is a set of records sharing the same key. For each group $\{(k, v_i)\}, 1 \leq i \leq n$, that correspond to the n records in the time window that have key k, an aggregate value $\hat{v} = v_1 \oplus v_2 \cdots \oplus v_n$ is computed, where $\oplus$ is an application-defined associative binary operator; e.g., `Sum`, `Max`, or `HyperLogLog` merge.[1] Customarily, the timeline is subdivided into non-overlapping intervals of length W and grouped aggregation is computed on each such interval.[2]

To compute windowed grouped aggregation, we consider aggregation at the edge as well as the center. The data records that arrive at the edge can be partially aggregated locally at the edge, so that the edge can maintain a set of partial aggregates, one for each

[1] More formally, $\oplus$ is any associative binary operator such that there exists a semigroup $(S, \oplus)$.
[2] Such non-overlapping time windows are often called *tumbling* windows in analytics terminology.

distinct key k. The edge may transmit, or *flush* these aggregates to the center; we refer to these flushed aggregates as *updates*. The center can further apply the aggregation operator $\oplus$ on incoming updates as needed in order to generate the final aggregate result. We assume that the computational overhead of the aggregation operator $\oplus$ is a small constant compared to the network overhead of transmitting an update.

Optimization Metrics

Our goal is to *simultaneously* minimize two metrics: *staleness*, a key measure of timeliness; and *network traffic*, a key measure of cost. Staleness is defined as the smallest time s such that the results of grouped aggregation for time window $[T, T+W)$ are available at the center at time $T+W+s$, as illustrated in Figure 4.3. In our model, staleness is simply the time elapsed from when the time window completes to when the last update for that time window reaches the center and is included in the final aggregate. The network traffic is measured by the number of updates sent over the network from the edge to the center.

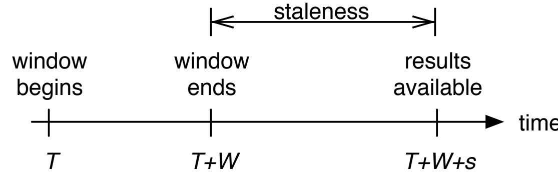


Figure 4.3: Staleness s is defined as the delay between the end of the window and final results for the window becoming available at the center.

Algorithms for Grouped Aggregation

An aggregation algorithm runs on the edge and takes as input the sequence of arrivals for data records in a given time window $[T, T+W)$. The algorithm produces as output a sequence of updates that are sent to the center. For each distinct key k with n_k arrivals in the time window, suppose that the ith data record $(k, v_{i,k})$ arrives at time $a_{i,k}$, where $T \leq a_{i,k} < T+W$ and $1 \leq i \leq n_k$. For each key k, the output of the aggregation algorithm is a sequence of m_k updates where the ith update $(k, \hat{v}_{i,k})$ departs[3] for the center at time $d_{i,k}$, $1 \leq i \leq m_k$. The updates must have the following properties:

[3] Upon departure from the edge, an update is handed off to the network for transmission.

- Each update for each key k aggregates all values for that key in the current time window that have not been previously included in an update.
- Each key k that has $n_k > 0$ arrivals must have $m_k > 0$ updates such that $d_{m_k,k} \geq a_{n_k,k}$. That is, each key with an arrival must have at least one update and the last update must depart after the final arrival so that all the values received for the key have been aggregated.

The goal of the aggregation algorithm is to minimize traffic, which is simply the total number of updates, i.e., $\sum_k m_k$. The other simultaneous goal is to minimize staleness, which is the time for the final update to reach the center, i.e., the time for the update with the largest value for $d_{m_k,k}$, to reach the center.[4]

4.3 Dataset and Workload

To derive a realistic workload for evaluating our aggregation algorithms, we have used anonymized workload traces obtained from a real-world analytics service[5] offered by Akamai, a large content delivery network. The download analytics service is used by content providers to track important metrics about who is downloading their content, from where is it being downloaded, what performance the users experienced, how many downloads completed, etc. The data source is a software called Download Manager that is installed on mobile devices, laptops, and desktops of millions of users around the world. The Download Manager is used to download software updates, security patches, music, games, and other content. The Download Manager instances installed on users' devices around the world send information about downloads to the widely-deployed edge servers using "beacons".[6] Each download results in one or more beacons being sent to an edge server containing information pertaining to that download. The beacons contain anonymized information about the time the download was initiated, its url, content size, number of bytes downloaded, user's ip, user's network, user's geography, server's network and server's geography. Throughout this thesis, we use the anonymized

[4] We implicitly assume a FIFO ordering of data records over the network, as is typically the case with protocols like TCP.

[5] `http://www.akamai.com/dl/feature_sheets/Akamai_Download_Analytics.pdf`

[6] A beacon is simply an http `GET` issued by the Download Manager for a small GIF containing the reported values in its url query string.

beacon logs from Akamai's download analytics service for the month of December, 2010. Note that we normalize derived values from the data set such as data sizes, traffic sizes, and time durations, for confidentiality reasons.

Throughout our evaluation, we compute grouped aggregation for three typical queries in the download analytics service. Queries that are issued in the download analytics service that use the grouped aggregation primitive can be roughly classified according to *query size*, which we define to be the number of distinct keys that are possible for that query. We choose three representative queries for different size categories (see Table 4.1). The `small` query groups by two dimensions with the key consisting of the tuple of content provider id and the user's last mile bandwidth classified into four buckets. The `medium` query groups by three dimensions with the key consisting of the triple of the content provider id, user's last mile bandwidth, and the user's country code. The `large` query groups by a different set of three dimensions with the key consisting of the triple of the content provider id, the user's country code, and the url accessed. Note that the last dimension—url—can take on hundreds of thousands of distinct values, resulting in a very large query size.

Table 4.1: Queries used throughout this thesis.

Name	Key	Value (aggregate type)	Description	Query Size
Small	`(cpid, bw)`	bytes downloaded (integer sum)	Total bytes downloaded by content provider by last-mile bandwidth.	$O(10^2)$ keys
Medium	`(cpid, bw, country_code)`	bytes downloaded (first 5 moments)	Mean and standard deviation of total bytes per download by content provider by bandwidth by country.	$O(10^4)$ keys
Large	`(cpid, bw, url)`	client ip (`HyperLogLog`)	Approximate number of unique clients by content provider by bw by url.	$O(10^6)$ keys

The total arrival rate of data records across all keys for all three queries is the same, since each arriving beacon contributes a data record for each query. However, the three queries have a different distribution of those arrivals across the possible keys, as shown in Figure 4.4(a). Recall that the large query has a large number of possible keys. About 56% of the keys for the large query arrive only once in the trace whereas the same percentage of keys for the medium and small query is 29% and 15% respectively. The

median arrival rate per key is four (resp., nine) times larger for the medium (resp., small) query in comparison with the large query. Figure 4.4(b) shows the number of unique keys arriving per hour at an edge server for the three queries. The figure shows the hourly and daily variations and also the variation across the three queries. Figure 4.4(c) shows the distribution of variation in interarrival times. Many keys have a coefficient of variation greater than 1, indicating relatively bursty arrivals.

4.4 Minimizing WAN Traffic and Staleness

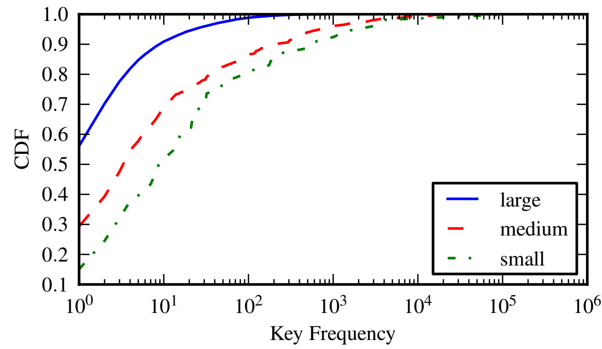
We now explore how to simultaneously minimize both traffic and staleness. We show that if the entire sequence of updates is known beforehand, then it is indeed possible to simultaneously achieve the optimal value for both traffic and staleness. While this offline solution cannot be implemented in practice, it serves as a baseline to which any online algorithm can be compared. Further, it characterizes the optimal solution that helps us develop the more sophisticated online algorithms that we present in Section 4.5.

Lemma 4.1 (Traffic optimality)**.** *In each time window, an algorithm is* traffic-optimal *iff it flushes exactly one update to the center for each distinct key that arrived in the window.*

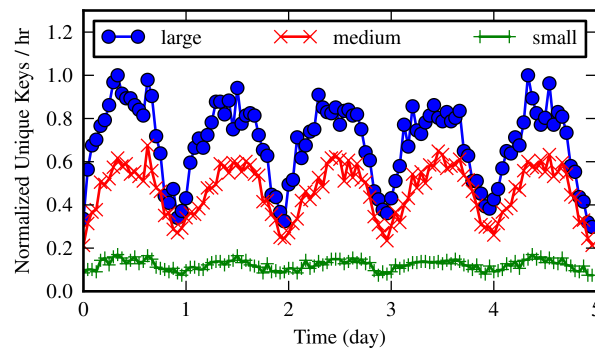
Proof. Any algorithm must flush at least one update for each distinct key that had arrivals in the time window. Suppose for contradiction that the algorithm flushes more than one update for a key. All flushes except the final one can be omitted, thereby decreasing the traffic, which is a contradiction. □

Intuitively, this lemma states that multiple records for each key within a window can be aggregated and sent as a single update to minimize traffic. Note that a pure batching algorithm satisfies the above lemma, and hence is traffic-optimal, but may not be staleness-optimal.

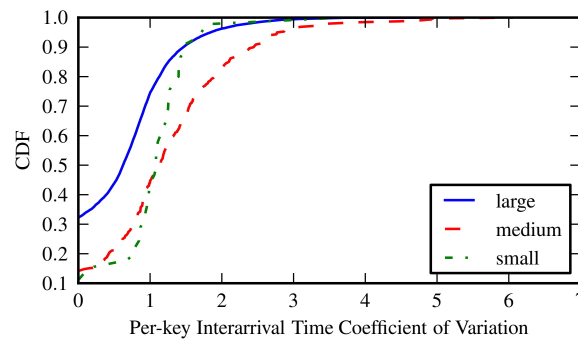
Lemma 4.2 (Staleness optimality)**.** *Let the optimal staleness for a time window* $[T, T+W)$ *be* S*. For any* $T \leq t < T+W$*, let* $N(t)$ *be the union of the set of keys that have outstanding updates (those not sent to the center yet) at time* t *and the set of keys that*



(a) CDF of the frequency per key at a single edge for the three queries.



(b) The unique key arrival rate for three different queries in a real-world web analytics service, normalized to the maximum rate for the large query.



(c) CDF of the coefficient of variation of per-key interarrival times, for keys with at least 25 arrivals.

Figure 4.4: Akamai Web download service data set characteristics.

arrive in $[t, T+W)$. *For a* staleness-optimal *algorithm the following holds:*

$$|N(t)| \leq \int_{t}^{T+W+S} b(\tau)d\tau, \forall T \leq t < T+W, \tag{4.1}$$

where $b(\tau)$ *is the instantaneous bandwidth at time* τ. *Further, if* $S > 0$ *there exists a critical time* t^* *such that the above Inequality 4.1 is satisfied with an equality.*

Proof. Inequality 4.1 holds since $N(t)$ records need to be flushed in interval $[t, T + W)$ and the maximum number of updates that can be sent before $T + W + S$ is $\int_{t}^{T+W+S} b(\tau)d\tau$. If $S > 0$, then let t^* be the time of arrival of the last key in the time window. If $|N(t^*)| < \int_{t^*}^{T+W+S} b(\tau)d\tau$, for some $S' < S$, $|N(t^*)| = \int_{t^*}^{T+W+S'} b(\tau)d\tau$, since there are no new arrivals after t^*. Thus, all updates for keys in $N(t^*)$ can be transmitted by time $T + W + S'$, decreasing the staleness to S', which is a contradiction. Hence, at time t^* Inequality 4.1 is satisfied with an equality. □

Intuitively, this lemma specifies that for a given arrival sequence, a staleness-optimal algorithm must send out pending updates to the center at a sufficient rate (dependent on the network bandwidth) to have them reach the center within the minimum feasible staleness bound.

Note that a pure streaming algorithm satisfies the above lemma if the network has sufficient capacity to stream *all arrivals* without causing network queues to fill. It need not, however, satisfy Lemma 4.1 if some of the groups have multiple arrivals within the window, and hence may not be traffic-optimal.

We now present *optimal offline algorithms* that minimize *both* traffic and staleness, provided the *entire* sequence of key updates is known to our aggregation algorithm *beforehand.*

Theorem 4.1 (Eager Optimal Algorithm). *There exists an optimal offline algorithm that schedules its flushes* eagerly*; i.e., it flushes exactly one update for each distinct key immediately after the last arrival for that key within the time window.*

Proof. Since the proposed algorithm flushes only a single update for each distinct key that arrived within the window, it is traffic-optimal as per Lemma 4.1. Clearly, any aggregation algorithm must flush an update for a key *after* the last arrival for that key, since the update must include the data contained in that last arrival in the final

aggregate for that window. Suppose there exists a key where the last arrival is at time t but the update to the center is sent at $t + \delta$, for some $\delta > 0$. Modifying that schedule such that the update is flushed at time t instead of $t+\delta$ cannot increase staleness. Thus, there exists an eager schedule that achieves the same staleness. □

Intuitively, this algorithm is traffic-optimal since it sends only one update per key, and is also staleness-optimal since it sends each update without any additional delay. We call the optimal offline algorithm described above the *eager optimal algorithm* due to the fact that it eagerly flushes updates for each distinct key immediately after the final arrival to that key. This eager algorithm is just one possible algorithm to achieve both minimum traffic and staleness. It might well be possible to delay flushes for some groups and still achieve optimal traffic and staleness. An extreme version of such an algorithm is a *lazy optimal algorithm* that flushes updates at the latest possible time that would still provide the optimal value of staleness and is described below.

1. Let keys $k_i, 1 \leq i \leq n$ have their last arrival at times $l_i, 1 \leq i \leq n$ respectively. Order the n keys that require flushing in the given window in the increasing order of their last arrival, i.e., the keys are ordered such that $l_1 \leq l_2 \leq \cdots l_n$.

2. Compute the minimum possible staleness S using the eager optimal algorithm.

3. As the base case, schedule the flush for the last key k_n at time $S - \delta_n$, where δ_n is the time required to transmit the update for k_n. That is, the last update is scheduled such that it arrives at the center with staleness exactly equal to S.

4. Now, iteratively schedule k_i, assuming all keys $k_j, j > i$ have already been scheduled. The update for k_i is scheduled at time $t_{i+1} - \delta_i$, where δ_i is the time required to transmit the update for k_i. That is, the update for k_i is scheduled such that the update for k_{i+1} is scheduled immediately after the update of k_i completes.

Theorem 4.2 (Lazy Optimal Algorithm)**.** *The lazy algorithm above is both traffic- and staleness-optimal.*

Proof. By construction. □

Further, consider a family of offline algorithms $\mathcal{A}$, where an algorithm $A \in \mathcal{A}$ schedules its update for key k_i at time t_i such that $e_i \leq t_i \leq l_i$, where e_i and l_i are the update times for key k_i in the eager and lazy schedules respectively. The following clearly holds.

Theorem 4.3 (Family of Offline Optimal Algorithms)**.** *Any algorithm $A \in \mathcal{A}$ is both traffic- and staleness-optimal.*

4.5 Practical Online Algorithms

In this section, we explore practical online algorithms for grouped aggregation, that strive to minimize both traffic and staleness. To ease the design of such online algorithms, we first frame the edge aggregation problem as an equivalent *caching problem.* This formulation has two advantages. First, it allows us to decompose the problem into two subproblems: determining the *cache size*, and defining a *cache eviction policy.* Second, as we will show, while the first subproblem can be solved by using insights gained from the optimal offline algorithms, the second subproblem lends itself to using the enormous prior work on cache replacement policies [62].

Concretely, we frame the grouped aggregation problem as a caching problem by treating the set of aggregates $\{(k_i, \hat{v}_i)\}$ maintained at the edge as a cache. A novel aspect of our formulation is that the *size of this cache changes dynamically.* Concretely, the cache works as follows:

- *Cache insertion* occurs upon the arrival of a record (k, v). If an aggregate with key k and value v_e exists in the cache (a "cache hit"), the cached value for key k is updated as $v \oplus v_e$ where $\oplus$ is the binary aggregation operator defined in Section 4.2. If no aggregate exists with key k (a "cache miss"), then (k, v) is added to the cache.

- *Cache eviction* occurs as the result of a cache miss when the cache is already full, or due to a *decrease* in the cache size. When an aggregate is evicted, it is flushed downstream and cleared from the cache.

Given the above definition of cache mechanics, we can express any grouped aggregation algorithm as an equivalent caching algorithm where the updates flushed by the aggregation algorithm correspond to evictions of the caching algorithm. More formally:

Theorem 4.4. *An aggregation algorithm A corresponds to a caching algorithm C such that:*

1. *At any time step, C maintains a cache size that equals the number of pending aggregates (those not sent to the center yet) for A, and*
2. *if A flushes an update for a key in a time step, C evicts the same key from its cache in that time step.*

Proof. By construction. □

Thus, any aggregation algorithm can be viewed as a caching algorithm with two policies: one for cache sizing and the other for cache eviction. In practice, we can define an online caching algorithm by defining online *cache size* and *cache eviction* policies. While the cache size policy determines *when* to flush updates, the cache eviction policy identifies *which* updates to flush at these times. Here we develop policies by attempting to emulate the behavior of the offline optimal algorithms using online information. We explore such online algorithms and the resulting tradeoffs in the rest of this section.

4.5.1 Simulation Methodology

To evaluate the relative merits of these algorithms, we implement a simple simulator in Python. Our simulator models each algorithm as a function that maps from arrival sequences to update sequences. Traffic is simply the length of the update sequence, while staleness is evaluated by modeling the network as a queueing system with deterministic service times, and arrival times determined by the update sequence. Note that we deliberately employ a simplified simulation, as the focus here is not on understanding performance in absolute terms, but rather to compare the tradeoffs between different algorithms.

We use these insights to develop practical algorithms that we implement in Apache Storm and deploy on PlanetLab (Section 4.6). Note that throughout the remainder of this section, we present results for the `large` query, but similar trends also apply to the `small` and `medium` queries.

4.5.2 Emulating the Eager Optimal Algorithm

Cache Size

To emulate the cache size corresponding to that for an eager offline optimal algorithm, we observe that, at any given time instant, an aggregate for key k_i is cached only if: in the window, (i) there has already been an arrival for k_i, and (ii) another arrival for k_i is yet to occur. We attempt to compute the number of such keys using two broad approaches: analytical and empirical.

In our analytical approach, the eager optimal cache size at a time instant can be estimated by computing the *expected* number of keys at that instant for which the above conditions hold. To compute this value, we model the arrival process of records for each key k_i as a Poisson process with mean arrival rate λ_i. Then the probability $p_i(t)$ that the key k_i should be cached at a time instant t within a window $[T, T+W)$ is given by $p_i(t) = 1 - \hat{t}^{W\lambda_i} - \left(1-\hat{t}\right)^{W\lambda_i}$, where $\hat{t} = (t-T)/W$.[7]

We consider two different models to estimate the arrival processes for different keys. The first model is a *Uniform* analytical model, which assumes that key popularities are uniformly distributed, and each key has the same mean arrival rate λ. Then, if the total number of keys arriving during the window is k, the expected number of cached keys at time t is simply $k \cdot \left(1 - \hat{t}^{W\lambda} - \left(1-\hat{t}\right)^{W\lambda}\right)$.

However, as Figure 4.4(a) in Section 4.3 demonstrated, key popularities in reality may be far from uniform. A more accurate model is the *Nonuniform* analytical model, that assumes each key k_i has its own mean arrival rate λ_i, so that the expected number of cached keys at time t is given by $\sum_{i=1}^{k} p_i\left(\hat{t}\right)$.

An online algorithm built around these models requires predicting the number of unique keys k arriving during a window as well as their arrival rates λ_i. In our evaluation, we use a simple prediction: assume that the current window resembles the prior window, and derive these parameters from the arrival history in the prior window.

Our empirical approach, referred to as *Eager Empirical*, also uses the history from the prior window as follows: apply the eager offline optimal algorithm to the arrival sequence from the previous window, and use the resulting cache size at time $t - W$ as

[7] Note that $W\lambda_i > 0$ since we are considering only keys with more than 0 arrivals, and that $\hat{t} < 1$ since $T \leq t < T + W$.

the prediction for cache size at time t.

Figure 4.5(a) plots the predicted cache size using these policies, along with the eager optimal cache size as a baseline. We observe that the Uniform model, unsurprisingly, is less accurate than the Nonuniform model. Specifically, it overestimates the cache size, as it incorrectly assumes that arrivals are uniformly distributed across many keys, rather than focused on a relatively small subset of relatively popular keys. Further, we see that the Eager Empirical model and the Nonuniform model both provide reasonably accurate predictions, but are prone to errors as the arrival rate changes from window to window.

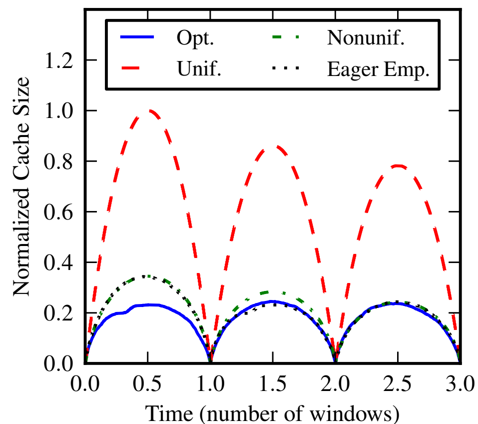
Cache Eviction

Having determined the cache size, the next issue is which keys to evict when needed. We know that an optimal algorithm will only evict keys without future arrivals. However, determining such keys accurately requires knowledge of the future. Instead, to implement a practical online policy, we consider two popular practical eviction algorithms—namely least-recently used (LRU), and least-frequently used (LFU)—and examine their interaction with the above cache size policies.

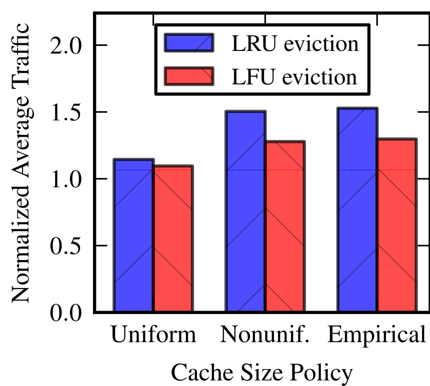
Figures 4.5(b) and 4.5(c) show the traffic and staleness, respectively, for different combinations of these cache size and cache eviction policies. Here, we simulate the case where network capacity is roughly five times that needed to support the full range of algorithms from pure batching to pure streaming. In these figures, traffic is normalized relative to the traffic generated by an optimal algorithm, while staleness is normalized by the window length.

From these figures, we see that the Eager Empirical and Nonuniform models yield similar traffic, though their staleness varies. It is worth noting that although the difference in staleness appears large in relative terms, the absolute values are still extremely low relative to the window length (less than 0.0015%), and are very close to optimal. We also see that LFU is the more effective eviction policy for this trace.

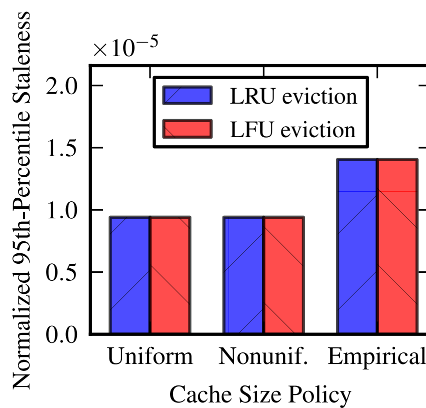
The more interesting result, however, is that the Uniform model, which produces the *worst* estimate of cache size, actually yields the *best* traffic: only about 9.6% higher than optimal, while achieving the same staleness as optimal. The reason is that, the more aggressive the cache size policy is in evicting keys prior to the end of the window,



(a) Cache sizes for emulations of the eager offline optimal algorithm.



(b) Traffic normalized relative to an optimal algorithm.



(c) Staleness normalized by window length.

Figure 4.5: Eager online algorithms.

the more pressure it places on an imperfect cache eviction algorithm to predict which key is least likely to arrive again.

On the other hand, when combined with the most accurate model of eager optimal cache size (Nonuniform), even the best practical eviction policy (LFU) generates 28% more traffic than optimal. This result indicates that leaving more headroom in the cache size (as done by Uniform) provides more robustness to errors by an online cache eviction policy.

4.5.3 Emulating the Lazy Optimal Algorithm

Cache Size

To emulate the lazy optimal offline algorithm (Section 4.4), we estimate the cache size by working backwards from the end of the window, determining how large the cache should be such that it can be drained by the end of the window (or as soon as possible thereafter) by fully utilizing the network capacity. This estimation must account for the fact that new arrivals will still occur during the remainder of the window, and each of those that is a cache miss will lead to an additional update in the future. This leads to a cache size $c(t)$ at time t defined as: $c(t) = \max\left(\bar{b} \cdot (T + W - t) - M(t), 0\right)$, where $\bar{b}$ denotes the average available network bandwidth for the remainder of the window, $T + W$ the end of the time window, and $M(t)$ the total number of cache misses that will occur during the remainder of the window.

Based on the above cache size function, an online algorithm needs to estimate the average bandwidth $\bar{b}$ and the number of cache misses $M(t)$ for the remainder of the window. We begin by focusing on the estimation of $M(t)$. We consider the bandwidth estimation problem in more detail in Section 4.5.4, and assume a perfect knowledge of $\bar{b}$ here. To estimate $M(t)$, we consider the following approaches. First, we can use a *Pessimistic* policy, where we assume that *all* remaining arrivals in the window will be cache misses. Concretely, we estimate $M(t) = \int_t^{T+W} a(\tau)\,\mathrm{d}\tau$ where $a(t)$ is the arrival rate at time t. In practice, this requires the prediction of the future arrival rate $a(t)$. In our evaluation, we simply assume that the future arrival rate is equal to the average arrival rate so far in the window.

Another alternative is to use an *Optimistic* policy, which assumes that the *current cache miss rate* will continue for the remainder of the window. In other words, $M(t) = m(t) \int_t^{T+W} a(\tau)\,\mathrm{d}\tau$ where $m(t)$ is the miss rate at time t. In our evaluation, we predict the arrival rate in the same manner as for the Pessimistic policy, and we use an exponentially weighted moving average for computing the recent cache miss rate.

A third approach is the *Lazy Empirical* policy, which is analogous to the Eager Empirical approach. It estimates the cache size by emulating the lazy offline optimal algorithm on the arrivals for the prior window.

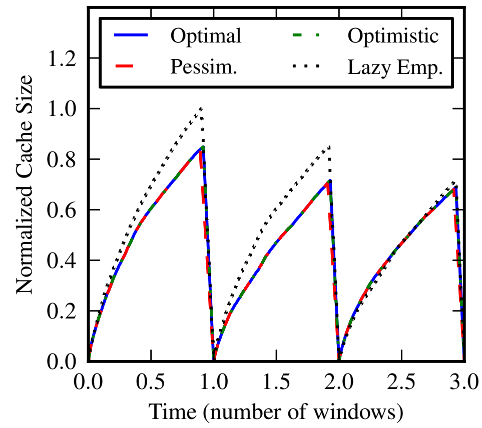
Figure 4.6(a) shows the cache size produced by each of these policies. We see that both the Lazy Empirical and Optimistic models closely capture the behavior of the optimal algorithm in dynamically decreasing the cache size near the end of the window. The Pessimistic algorithm, by assuming that all future arrivals will be cache misses, decays the cache size more rapidly than the other algorithms.

Cache Eviction

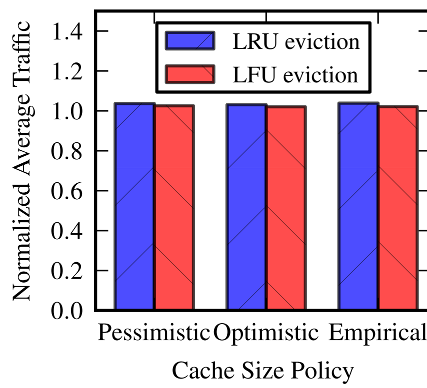
We explore the same eviction algorithms here, namely LRU and LFU, as we did in Section 4.5.2. Figures 4.6(b) and 4.6(c) show the traffic and staleness, respectively, generated by different combinations of these cache size and cache eviction policies. We see that LFU again slightly outperforms LRU. More importantly, we see that, regardless of which cache size policy we use, these lazy approaches outperform the best online eager algorithm in terms of traffic. Even the worst lazy online algorithm produces traffic less than 4% above optimal.

The results for staleness, however, show a significant difference between the different policies. We see that by assuming that all future arrivals will be cache misses, the Pessimistic policy achieves enough tolerance in the cache size estimation, avoiding overloading the network towards the end of the window, and leading to low staleness.

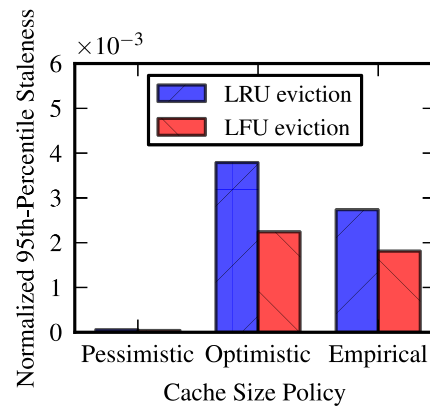
Based on the results so far, we see that accurately modeling the optimal cache size does not yield the best results in practice. Instead, our algorithms should be lazy, deferring updates until later in the window, and in choosing how long to defer, they should be pessimistic in their assumptions about future arrivals.



(a) Cache sizes for emulations of the lazy offline optimal algorithm.



(b) Traffic normalized relative to an optimal algorithm.



(c) Staleness normalized by window length.

Figure 4.6: Lazy online algorithms.

4.5.4 The Hybrid Algorithm

In the discussion of the lazy online algorithm, we assumed perfect knowledge of the future network bandwidth $\bar{b}$. In practice, however, if the actual network capacity turns out to be lower than the predicted value, then network queueing delays may grow, potentially resulting in high staleness.

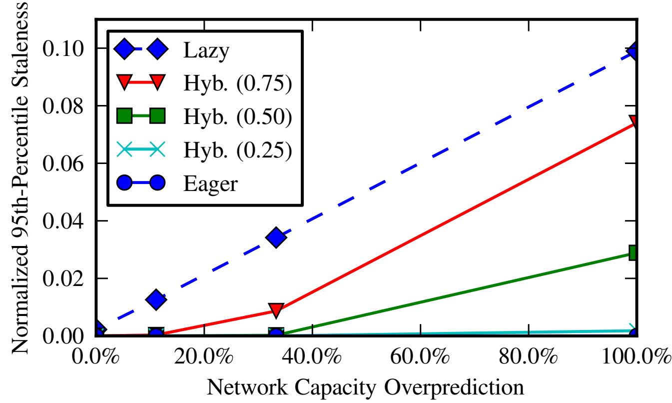


Figure 4.7: Sensitivity of the hybrid algorithms with a range of α values to overpredicting the available network capacity. Staleness is normalized by window length.

Figure 4.7 shows how staleness increases as the result of overpredicting network capacity. Note that the predicted capacity remains constant, while we vary the actual network capacity. The top-most curve corresponds to a lazy online algorithm (Pessimistic + LFU) which is susceptible to very high staleness if it overpredicts network capacity (up to 9.9% of the window length for 100% overprediction).

To avoid this problem, recall Theorem 4.3, where we observed that the eager and lazy optimal algorithms are merely two extremes in a family of optimal algorithms. Figure 4.8 shows, for three windows, the size of the cache for both eager and lazy optimal algorithms. This figure shows that the eager algorithm maintains a smaller cache size than the lazy algorithm, as lazy retains some keys long after their last arrival. This indicates that there is flexibility in terms of the cache size between these two extremes (those for eager and lazy). Further, our results from Sections 4.5.2 and 4.5.3 showed that it is useful to add headroom to the accurate cache size estimates: towards a larger (resp., smaller) cache size in case of the eager (resp., lazy) algorithm. These

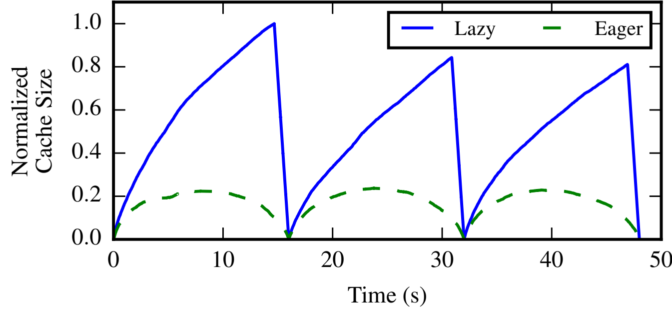


Figure 4.8: Cache size over time for eager and lazy offline optimal algorithms. Sizes are normalized relative to the largest size.

insights indicate that a more effective cache size estimate should lie somewhere between the estimates for the eager and lazy algorithms.

We therefore propose a *Hybrid* algorithm that computes cache size as a linear combination of eager and lazy cache sizes. Concretely, a Hybrid algorithm with a *laziness parameter* α—denoted by Hybrid(α)—estimates the cache size $c(t)$ at time t as: $c(t) = \alpha \cdot c_l(t) + (1 - \alpha) \cdot c_e(t)$, where $c_l(t)$ and $c_e(t)$ are the lazy and eager cache size estimates, respectively. In our evaluation, we use the Nonuniform model for the eager and the Optimistic model for the lazy cache size estimation respectively, as these most accurately capture the cache sizes of their respective optimal baselines.

Observing Figure 4.7 again, we see that as we decrease the laziness parameter (α) below about 0.5, and use a more eager approach, the risk of bandwidth misprediction is largely mitigated, and the staleness even under significant bandwidth overprediction remains small.

Note that since predicted network capacity is constant in this figure, traffic is fixed for each algorithm irrespective of the bandwidth prediction error. Figure 4.9 shows that as we use a more eager hybrid algorithm, traffic increases. This illustrates a tradeoff between traffic and staleness in terms of achieving robustness to network bandwidth overprediction. A reasonable compromise seems to be a low α value, say 0.25. Using this algorithm, traffic is less than 6.0% above optimal, and even when network capacity is overpredicted by 100%, staleness remains below 0.19% of the window length.

Overall, we find that a purely eager online algorithm is susceptible to errors by practical eviction policies, while a purely lazy online algorithm is susceptible to errors in

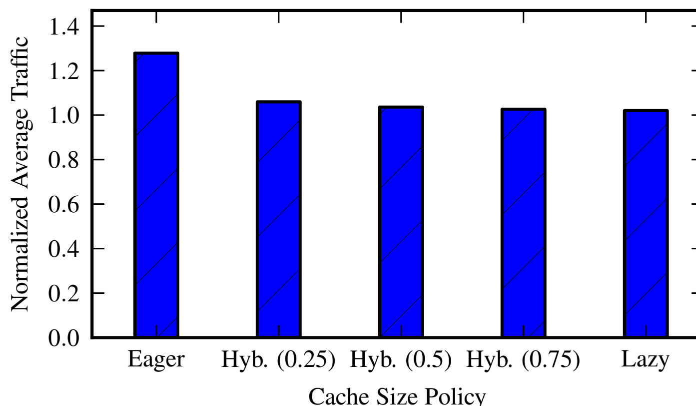


Figure 4.9: Average traffic for hybrid algorithms with several values of the laziness parameter α. Traffic is normalized relative to an optimal algorithm.

bandwidth prediction. A hybrid algorithm that combines these two approaches provides a good compromise by being more robust to errors in both arrival process and bandwidth estimation.

4.6 Implementation

We demonstrate the practicality of our algorithms and ultimately their performance by implementing them in Apache Storm [58]. Our prototype uses a distinct Storm cluster at each edge, as well as at the center, in order to distribute the work of aggregation. We choose this multi-cluster approach rather than attempting to deploy a single geo-distributed Storm cluster for two main reasons. First, a single global Storm cluster would require a custom task scheduler in order to control task placement. Second, and much more critically, Storm was designed and has been optimized for high performance within a single datacenter; it would not be reasonable to expect it to perform well in a geo-distributed setting characterized by high latency and high degrees of compute and bandwidth heterogeneity.

Figure 4.10 shows the overall architecture, including the edge and center Storm topologies. We briefly discuss each component in the order of data flow from edge to center.

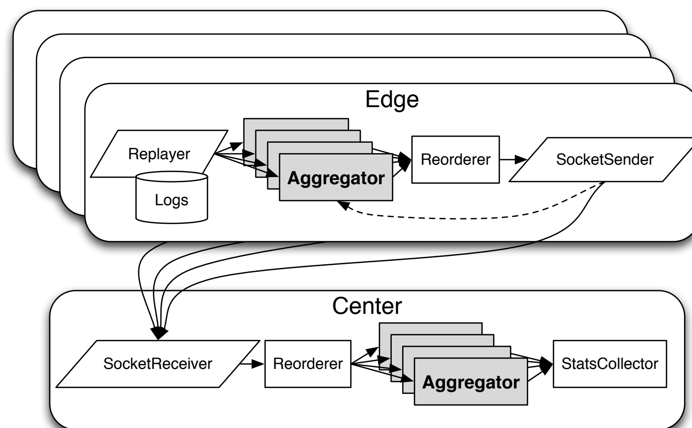


Figure 4.10: Aggregation is distributed over Apache Storm clusters at each edge as well as at the center.

4.6.1 Edge

Data enters our prototype at the edge through the `Replayer` spout, which replays timestamped logs from a file, and can speed up or slow down log replay to explore different stream arrival rates. Each line is parsed using a query-specific parsing function to produce a `(timestamp, key, value)` triple. Our implementation supports a broad set of value types and associated aggregations by leveraging Twitter's Algebird[8] library. The `Replayer` emits records according to their timestamp; i.e., event time and processing time [15] are equivalent at the `Replayer`. The `Replayer` emits records downstream, and also periodically emits punctuation messages to indicate that no messages with earlier timestamps will be sent in the future.

The next step in the dataflow is the `Aggregator`, for which one or more tasks run at each cluster. The `Aggregator` defines window boundaries in terms of record timestamps, and maintains the dynamically sized cache from Section 4.5, with each task aggregating a hash-partitioned subset of the key space. We generalize over a broad range of *eviction policies* by ordering keys using a priority queue with an efficient `changePriority` implementation. By defining priority as a function of key, value, existing priority (if any) and the time that the key was last updated in the map, we can capture a broad range of algorithms including LRU and LFU.

[8] `https://github.com/twitter/algebird`

The `Aggregator` also maintains a cache size function, which maps from time within the window to a cache size. This function can be changed at runtime in order to support implementing arbitrary dynamic sizing policies. This mechanism can be used, for example, to update the cache size function based on arrival rate and miss rate as in our Lazy Pessimistic algorithm, or to record the arrival history for one window and use this history to compute the size function for the next window, as in the Eager Empirical algorithm.[9]

The `Aggregator` tasks send their output to a single instance of the `Reorderer` bolt, which is responsible for delaying records as necessary in order to maintain punctuation semantics. Data then flows into the `SocketSender` bolt, which transmits partial aggregates to the center using TCP sockets. This `SocketSender` also maintains an estimate of network bandwidth to the center, and periodically emits these estimates upstream to `Aggregator` instances for use in defining their cache size functions. Our bandwidth estimation is based on simple measurements of the rate at which messages can be sent over the network. For a more reliable prediction, we could employ lower-level techniques [63], or even external monitoring services [49].

4.6.2 Center

At the center, data follows largely the reverse order. First, the `SocketReceiver` spout is responsible for deserializing partial aggregates and punctuations and emitting them downstream into a `Reorderer`, where the streams from multiple edges are synchronized. From there, records flow into the central `Aggregator`, each task of which is responsible for performing the final aggregation over a hash-partitioned subset of the key space. Upon completing aggregation for a window, these central `Aggregator` tasks emit summary metrics including traffic and staleness, and these metrics are summarized by the final `StatsCollector` bolt. Staleness is computed relative to the wall-clock (i.e., processing) time at which the window closes. Clocks are synchronized using NTP.[10]

[9] For our experiments, we use this mechanism to implement a baseline cache size policy that learns the eager optimal eviction schedule after processing the log trace once.

[10] Clock synchronization errors are small relative to the staleness ranges we explore in our experiments.

Note that our prototype achieves at-most-once delivery semantics. Storm's acknowledgment mechanisms can be used to implement at-least-once semantics, and exactly-once semantics can be achieved by employing additional checks to filter duplicate updates, though we have not implemented these measures.

4.7 Experimental Evaluation

To evaluate the performance of our algorithms in a real geo-distributed setting, we deploy our Apache Storm architecture on a PlanetLab testbed. Our PlanetLab deployment uses a total of eleven nodes (64 total cores) spanning seven sites. Central aggregation is performed using a Storm cluster at a single node at `princeton.edu`.[11] Edge locations include `csuohio.edu`, `uwaterloo.ca`, `yale.edu`, `washington.edu`, `ucla.edu`, and `wisc.edu`. Bandwidth from edge to center varies from as low as 4.5 Mbps (`csuohio.edu`) to as high as 150 Mbps (`yale.edu`), based on `iperf`. To emulate streaming data, each edge replays a geographic partition of the CDN log data described in Section 4.3. To explore the performance of our algorithms under a range of workloads, we use the three diverse queries described in Table 4.1, and we replay the logs at both low and high (8x faster than low) rates. Note that for confidentiality purposes, we do not disclose the actual replay rates, and we present staleness and traffic results normalized relative to the window length and optimal traffic, respectively.

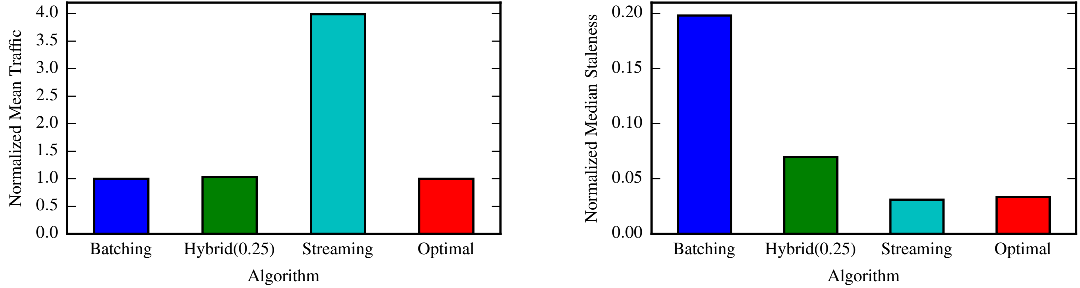
4.7.1 Aggregation Using a Single Edge

Although our work is motivated by the general case of multiple edges, our algorithms were developed based on an in-depth study of the interaction between a single edge and center. We therefore begin by studying the real-world performance of our hybrid algorithm when applied at a single edge. Following the rationale from Section 4.5.4, we choose a laziness parameter of $\alpha = 0.25$ for this initial experiment, though we will study the tradeoffs of different parameter values shortly.

Compared to the extremes of pure batching and pure streaming, as well as an optimal algorithm based on a priori knowledge of the data stream, our algorithm performs quite

[11] We originally employed multiple nodes at the center, but were forced to confine our central aggregation to a single node due to PlanetLab's restrictive limitations on daily network bandwidth usage that was quickly exhausted by the communication between Storm workers.

well. Figures 4.11(a) and 4.11(b) show that our hybrid algorithm very effectively exploits the opportunity to reduce bandwidth relative to streaming, yielding traffic less than 2% higher than the optimal algorithm. At the same time, our hybrid algorithm is able to reduce staleness by 65% relative to a pure batching algorithm.



(a) Mean traffic (normalized relative to an optimal algorithm).

(b) Median staleness (normalized by window length).

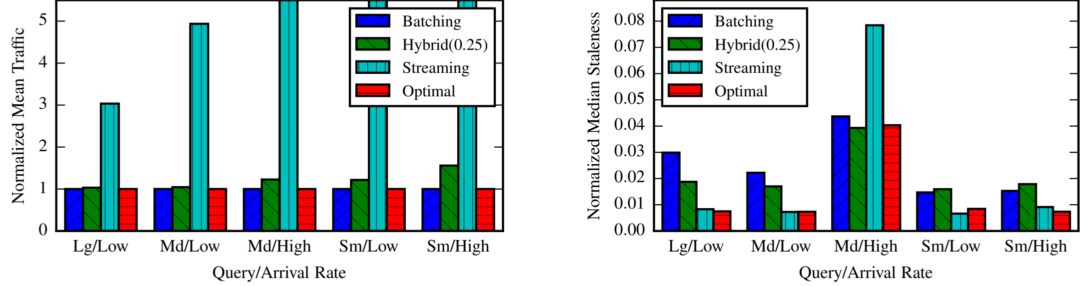
Figure 4.11: Performance for batching, streaming, optimal, and our hybrid algorithm for the `large` query with a low stream arrival rate using a one-edge Apache Storm deployment on PlanetLab.

4.7.2 Scaling to Multiple Edges

Now, in order to understand how well our algorithm scales beyond a single edge, we partition the log data over three geo-distributed edges. We replay the logs at both low and high rates, and for each of the `large`, `medium`, and `small` queries.[12] As Figures 4.12(a) and 4.12(b) demonstrate, our hybrid algorithm performs well throughout. It is worth noting that the edges apply their cache size and cache eviction policies based purely on local information, without knowledge of the decisions made by the other edges, except indirectly via the effect that those decisions have on the available network bandwidth to the edge.

Performance is generally more favorable for our algorithm for the `large` and `medium` queries than for the `small` query. The reason is that, for these larger queries, while edge aggregation reduces communication volume, there is still a great deal of data to transfer from the edges to the center. Staleness is quite sensitive to precisely when these partial

[12] We do not present the results for `large`-high because the amount of traffic generated in these experiments could not be sustained within the PlanetLab bandwidth limits.



(a) Mean traffic (normalized relative to an optimal algorithm). Normalized traffic values for streaming are truncated, as they range as high as 164.

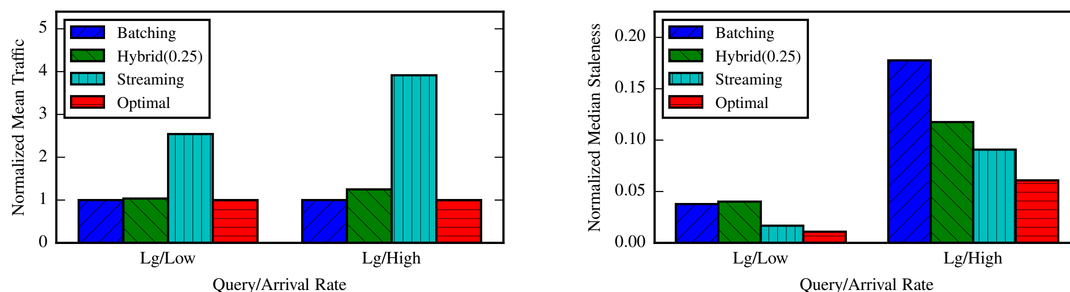
(b) Median staleness (normalized by window length).

Figure 4.12: Performance for batching, streaming, optimal, and our hybrid algorithm for a range of queries and stream arrival rates using a three-edge Apache Storm deployment on PlanetLab.

aggregates are transferred, and our algorithms work well in scheduling this communication. For the `small` query, on the other hand, edge aggregation is extremely effective in reducing data volumes, so much so that there is little risk in delaying communication until the end of the window. For queries that aggregate extremely well, batching is a promising algorithm, and we do not necessarily outperform batching. The advantage of our algorithm over batching is therefore its broader applicability: the Hybrid algorithm performs roughly as well as batching for small queries, and significantly outperforms it for large queries.

We continue by further partitioning the log data across a total of six geo-distributed edges. Given the higher aggregate compute and network capacity of this deployment, we focus on the `large` query at both low and high arrival rates. From Figure 4.13(a), we can again observe that our hybrid algorithm yields near-optimal traffic. We can also observe an important effect of stream arrival rate: all else equal, a high stream arrival rate lends itself to more thorough aggregation at the edge. This is evident in the higher normalized traffic for streaming with the high arrival rate than with the low arrival rate.

In terms of staleness, Figure 4.13(b) shows that our algorithm performs well when the arrival rate is high and the network capacity is relatively constrained. In this case, staleness is more sensitive to the particular scheduling algorithm. When the arrival rate is low, we see that our hybrid algorithm performs slightly worse than batching,



(a) Mean traffic (normalized relative to an optimal algorithm).

(b) Median staleness (normalized by window length).

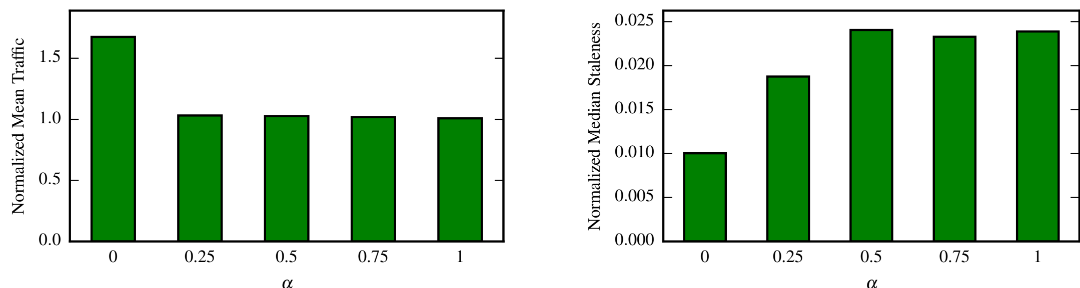
Figure 4.13: Performance for batching, streaming, optimal, and our hybrid algorithm for the `large` query with low and high stream arrival rates using a six-edge Apache Storm deployment on PlanetLab.

though in absolute terms the difference is quite small. Our hybrid algorithm generates higher staleness than streaming, but does so at a much lower traffic cost. Just as with the three-edge case, we again see that, where a large opportunity exists, our algorithm exploits it. Where an extreme algorithm such as batching already suffices, our algorithm remains competitive.

4.7.3 Effect of Laziness Parameter

In Section 4.5.4, we observed that a purely eager algorithm is vulnerable to mispredicting which keys will receive further arrivals, while a purely lazy algorithm is vulnerable to overpredicting network bandwidth. This motivated our hybrid algorithm, which uses a linear combination of eager and lazy cache size functions. We now explore the real-world tradeoffs of using a more or less lazy algorithm by running experiments with the `large` query at a low replay rate over three edges with laziness parameter α ranging from 0 through 1.0 by steps of 0.25. As expected based on our simulation results, Figure 4.14(a) shows that α has little effect on traffic when it exceeds about 0.25. Somewhere below this value, the imperfections of practical cache eviction algorithms (LRU in our implementation) begin to manifest. More specifically, at $\alpha = 0$, the hybrid algorithm reduces to a purely eager algorithm, which makes eviction decisions well ahead of the end of the window, and often chooses the wrong victim. By introducing even a small amount of laziness, say with $\alpha = 0.25$, this effect is largely mitigated.

Figure 4.14(b) shows the opposite side of this tradeoff: a lazier algorithm runs a higher risk of deferring communication too long, in turn leading to higher staleness. Based on staleness alone, a more eager algorithm is better. Based on the shape of these trends, $\alpha = 0.25$ appears to be a good compromise value for our experiments.



(a) Mean traffic (normalized relative to an optimal algorithm).

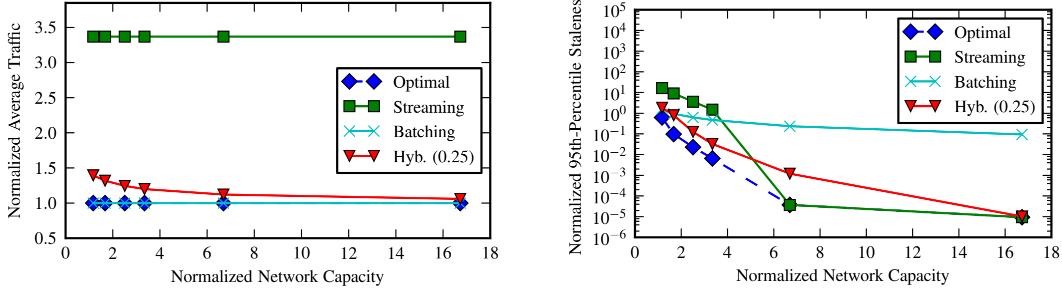
(b) Median staleness (normalized by window length).

Figure 4.14: Effect of laziness parameter α using a three-edge Apache Storm deployment on PlanetLab with query `large`.

4.7.4 Impact of Network Capacity

For our final set of results, we use simulations to fully understand the impact of network capacity on our algorithms. We use a simulation methodology for these results since PlanetLab gives us limited control over varying bandwidth capacities, and also has a maximum daily bandwidth cap. Here, we simulate these algorithms across a wide range of network capacities, ranging from highly constrained (less than 20% greater than optimal traffic) to highly unconstrained (about 5x more than that needed to support pure streaming). Figures 4.15(a) and 4.15(b) show traffic and staleness, respectively, for the four algorithms over this range of network capacities for the `large` query.

In terms of traffic, we see that our Hybrid(0.25) algorithm comes close to the optimal traffic, especially at higher network capacities. Traffic in the highly constrained regime is slightly worse than optimal, but still provides a significant improvement over that from pure streaming. The reason for this trend is that, as the network becomes more constrained, the envelope between lazy and eager algorithms shrinks, so that the hybrid algorithm has lower room for error. Note that batching is traffic-optimal, as discussed in Section 4.4.



(a) Traffic (normalized relative to an optimal algorithm)

(b) Staleness (normalized by window length). Y-axis is log-scale.

Figure 4.15: Traffic and staleness for different algorithms over a range of network capacities.

In terms of staleness, we see that streaming is nearly staleness-optimal at high network capacity, yet performs worse than all other alternatives under low network capacity. This is because, under a highly constrained network, excessive traffic from streaming leads to large—even unbounded—network queuing delays. Batching, on the other hand, is the worst alternative at high network capacity, yet yields near-optimal staleness at low network capacity. This is because it always defers communication until the end of the window, which can lead to high delay when the network is not a bottleneck, but on the other hand it prevents queue buildups under severe bandwidth constraints. Our Hybrid algorithm follows the same trend as the optimal, performing close to optimal irrespective of the network capacity.

4.8 Discussion

4.8.1 Compression

Rather than flushing individual key-value pairs, compression could be applied to batches of key-value pairs to reduce WAN traffic. Whether this is worthwhile, however, depends on the speed of compression and decompression relative to the WAN bandwidth, as well as the extent to which compression, transmission, and decompression can be pipelined. For example, if compression or decompression throughput is lower than the WAN bandwidth, then compression will become a bottleneck and only increase staleness. In the

wide-area settings motivating this work, however, WAN bandwidth is likely to remain the bottleneck.

Further, if compression is only effective over very large batches, then it may lead to higher rather than lower staleness. More specifically, as we increase the size of the batches to which we apply compression, the compression ratio increases, but so does the additional batching delay. In practice, however, it is possible to achieve a good compression ratio over reasonably small batches, and these batches can be compressed and decompressed quickly.

For example, we apply Google's Snappy compression [64] to our anonymized Akamai trace at several granularities, and show the results in Figure 4.16. We run each experiment five times and show the mean, with error bars indicating 95% confidence intervals for latency (compression ratio does not vary from run to run). We see that it is possible to achieve a high compression ratio over batches small enough to be compressed with sub-millisecond latency. In other words, compression can be effective in reducing WAN traffic even when applied at a fine enough granularity to avoid significantly increasing staleness.

Incorporating compression into our techniques is therefore quite promising, though implementing it in our experimental prototype remains an item for future work.

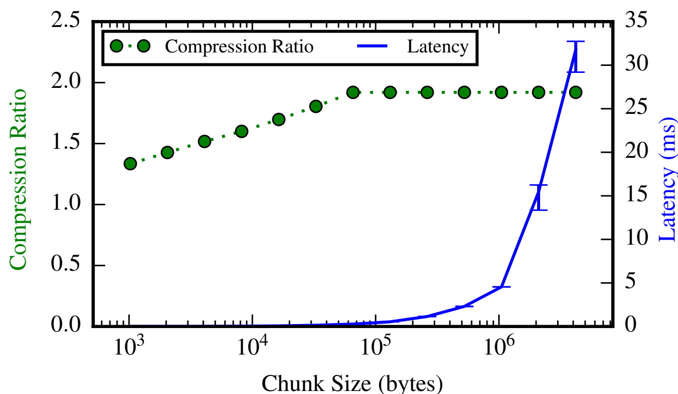


Figure 4.16: Compression ratio and latency for Google Snappy compression applied at several granularities to our anonymized Akamai trace.

4.8.2 Variable-Size Aggregates

We have assumed that the time to perform the binary aggregation operation is small relative to the time to transmit an aggregate over the WAN. This is a reasonable assumption in practice. Many aggregation operators (e.g., `Sum`, `Max`, `Average`) involve simple, lightweight computation. For more complex aggregations such as `HyperLogLog`, we assume that sufficient edge resources are provisioned so that computation is not the bottleneck. For situations where this is not the case, it is necessary to apply load-shedding techniques at the edge, though we do not study such techniques in this thesis.

We have also assumed that aggregate values have constant size. In practice, this is true for many aggregates; e.g., a `Sum` implemented using an 8-byte long integer has the same size regardless of its value. Even for many sophisticated aggregates, this assumption is reasonable, though there are opportunities for optimization. For example, consider an approximate set cardinality counter implemented using a `HyperLogLog` sketch [65]. The size of a `HyperLogLog` instance is dictated not by the set that it approximates, but by the error-tolerance parameter specified at its construction, which dictates the size of the underlying array of "registers". For large sets, this fixed-size representation is much smaller than an exact representation, but for small sets, the exact representation may be more concise.[13]

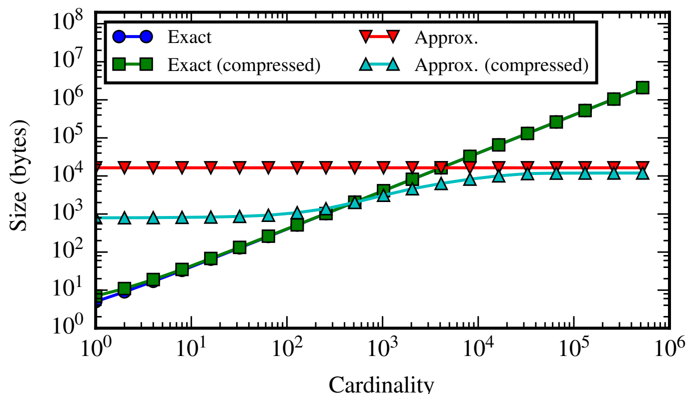


Figure 4.17: Size of exact and approximate set representations with and without compression.

[13] At intermediate cardinalities, a sparse representation of the register array may be most efficient.

As a concrete example, Figure 4.17 shows the space required to represent a set of random ip addresses using both exact and approximate representations with and without (Google Snappy) compression. Across a range of cardinalities, the approximate representation requires nearly constant space. Compression is useful for the approximate representation, especially at low cardinalities where the register array is sparse. At high cardinalities, the utility of the `HyperLogLog` is obvious: even an uncompressed representation is orders of magnitude more concise than an exact representation. At low cardinalities, however, the opposite is true: an exact representation is smaller than even a compressed approximate representation. In practice, choosing the best representation based on the given value is a powerful optimization.

Although we have not explored this optimization in our simulation or experiments, it is fundamentally compatible with our algorithms. We expect practical implementations to choose the most concise representation for a given aggregation (e.g., an exact set at low cardinalities, and a compressed `HyperLogLog` at high cardinalities). We then view our dynamic sizing policy as reflecting not the number, but rather the total size, of aggregates to allow in the cache. The eviction policy remains responsible for identifying which key is most likely to have attained its final aggregate value, a decision that is independent of the representation of the value for that key. This optimization allows more keys to remain cached at the edge at any given time, reducing the pressure to evict early. This reduces the rate of premature flushes, leading to lower traffic and staleness.

4.8.3 Applicability to Other Environments

While our focus is on geo-distributed stream analytics, our algorithms are applicable to other environments such as single data center or cluster environments. For instance, these algorithms could be used to achieve the desired metrics if an intra-data center network is the bottleneck, e.g., for communication across racks. Our algorithms could also be applicable to scheduling I/O to other devices (e.g., scheduling writes to disks) if they become the bottleneck instead of the network.

Although our work is motivated by the typical hub-and-spoke topology, our techniques are not confined to this model. They can also be applied to minimize traffic and staleness for deployments with multiple central sites, or in hierarchical topologies.

4.9 Related Work

Aggregation

Aggregation is a key operator in analytics, and grouped aggregation is supported by many data-parallel programming models [59, 66, 67]. Larson et al. [68] explore the benefits of performing partial aggregation prior to a join operation, much as we do prior to network transmission. While they also recognize similarities to caching, they consider only a fixed-size cache, whereas our approach uses a dynamically varying cache size. In sensor networks, aggregation is often performed over a hierarchical topology to improve energy efficiency and network longevity [69, 70], whereas we focus on cost (traffic) and timeliness (staleness). Amur et al. [71] study grouped aggregation, focusing on the design and implementation of efficient data structures for batch and streaming computation. They discuss tradeoffs between eager and lazy aggregation, but do not consider the effect on staleness, a key performance metric in our work.

Streaming Systems

Numerous streaming systems [58, 60, 72, 73, 74, 75, 76, 77] have been proposed over the years. These systems provide many useful ideas for new analytics systems to build upon, but they do not fully explore the challenges addressed here, in particular how to achieve timely results (low staleness) at low cost (low traffic).

Google recently proposed the Dataflow model [15] as a unified abstraction for computing over both bounded and unbounded datasets. Our work fits within this model. We focus in particular on aggregation over tumbling windows on unbounded data streams.

Wide-Area Computing

Wide-area computing has received increased research attention in recent years, due in part to the widening gap between data processing and communication costs. Much of this attention has been paid to batch computing [8, 20, 78]. Relatively little work on streaming computation [76] has focused on wide-area deployments, or associated questions such as where to place computation. Pietzuch et al. [79] optimize operator placement in geo-distributed settings to balance between system-level bandwidth usage and latency. Hwang et al. [80] rely on replication across the wide area in order to

achieve fault tolerance and reduce straggler effects. JetStream [12] considers wide-area streaming computation, but unlike our work, assumes that it is always better to push more computation to the edge.

Optimization Tradeoffs

LazyBase [81] provides a mechanism to trade increased staleness for faster query response in the case of ad-hoc queries. BlinkDB [82] and JetStream [12] provide mechanisms to trade accuracy for response time and bandwidth utilization, respectively. We focus on *jointly* optimizing both network traffic and staleness. Das et al. [83] consider tradeoffs between throughput and latency in Spark Streaming, but they consider only a uniform batching interval for the entire stream, while we address scheduling on a per-key basis.

4.10 Concluding Remarks

In this chapter, we focused on optimizing the important primitive of windowed grouped aggregation in a geo-distributed setting by minimizing two key metrics: WAN traffic, and staleness. We presented a family of optimal offline algorithms that *jointly minimize both staleness and traffic*. Using this as a foundation, we developed practical online aggregation algorithms based on the observation that grouped aggregation can be modeled as a *caching problem* where the cache size varies over time. We explored a range of online algorithms ranging from eager to lazy in terms of how early they send out updates. We found that a hybrid online algorithm works best in practice, as it is robust to a wide range of network constraints and estimation errors. We demonstrated the practicality of our algorithms through an implementation in Apache Storm, deployed on the PlanetLab testbed. The results of our experiments, driven by workloads derived from anonymized traces of Akamai's download analytics service, showed that our online aggregation algorithms perform close to the optimal algorithms for a variety of system configurations, stream arrival rates, and query types.

Chapter 5

Trading Timeliness and Accuracy in Geo-Distributed Streaming Analytics

5.1 Introduction

In Chapter 4, we examined questions of where to compute and when to communicate in the context of geo-distributed streaming analytics. We presented the design of algorithms for performing windowed grouped aggregation in order to optimize the two key metrics of WAN traffic and staleness. In doing so, we assumed that (a) applications required exact results, and (b) resources—WAN bandwidth in particular—were sufficient to deliver exact results. In general, however, these assumptions do not always hold. For one, it is not always feasible to compute exact results with bounded staleness [12]. Further, many real-world applications can tolerate some staleness or inaccuracy in their final results, albeit with diverse preferences. For instance, a network administrator may need to be alerted to potential network overloads *quickly* (within a few seconds or minutes) even if there is some degree of error in the results describing network load. On the other hand, a Web analyst might have only a small tolerance for error (say, <1%) in the application statistics (e.g., number of page hits) but be willing to wait for some time to obtain these results with the desired accuracy.

In this chapter, we study the *staleness-error tradeoff*, recognizing that applications have diverse requirements: some may tolerate higher staleness in order to achieve lower error, and vice versa. We devise both theoretically optimal offline algorithms as well as practical online algorithms to solve two complementary problems: minimize staleness under an error constraint, and minimize error under a staleness constraint. Our algorithms enable geo-distributed streaming analytics systems to support a diverse range of application requirements, whether WAN capacity is plentiful or highly constrained.

Research Contributions

- We study the tradeoff between staleness and error (measures of timeliness and accuracy, respectively) in a geo-distributed stream analytics setting.
- We present optimal offline algorithms that allow us to optimize staleness (resp., error) under an error (resp., staleness) constraint.
- Using these offline algorithms as references, we present practical online algorithms to efficiently trade off staleness and error. These practical algorithms are based on the key insight of representing grouped aggregation at the edge as a *two-part cache*. This formulation generalizes our caching-based framework for exact windowed grouped aggregation (Chapter 4) by introducing cache *partitioning* policies to identify which partial results must be sent and which can be discarded.
- We demonstrate the practicality and efficacy of these algorithms through both trace-driven simulations and implementation in Apache Storm [58], deployed on a PlanetLab [14] testbed. Using workloads derived from traces of a popular web analytics service offered by Akamai [9], a large commercial content delivery network, our experiments show that our algorithms reduce staleness by 81.8% to 96.6%, and error by 83.4% to 99.1% compared to a practical random sampling/batching-based aggregation algorithm.
- We demonstrate that our techniques apply across a diverse set of aggregates, from distributive and algebraic aggregates [66] such as `Sum` and `Max` to holistic aggregates such as unique count (via `HyperLogLog` [65]).

5.2 Problem Formulation

5.2.1 Problem Statement

We consider the same hub-and-spoke system model described in Section 4.2. Additionally, we begin with the same notion of windowed grouped aggregation. The primary difference in this chapter, however, is that we admit approximation.

Approximate Windowed Grouped Aggregation

An aggregation algorithm runs on the edge and takes as input the sequence of arrivals of data records in a given time window $[T, T+W)$. The algorithm produces as output a sequence of updates that are sent to the center.

For each distinct key k with $n_k > 0$ arrivals in the time window, suppose that the ith data record $(k, v_{i,k})$ arrives at time $a_{i,k}$, where $T \leq a_{i,k} < T+W$ and $1 \leq i \leq n_k$. For each such key k, the output of the aggregation algorithm is a sequence of m_k updates, where $0 \leq m_k \leq n_k$. The jth update $(k, \hat{v}_{j,k})$ departs for the center at time $d_{j,k}$ where $1 \leq j \leq m_k$. This update aggregates all values for key k that have arrived but have not yet been included in an update.

The final, possibly approximate, aggregated value for key k is then given by $\hat{V}_k = \hat{v}_{1,k} \oplus \hat{v}_{2,k} \oplus \cdots \oplus \hat{v}_{m_k,k}$. Approximation arises in two cases. The first is when, for a key k with $n_k > 0$ arrivals, no update is flushed; i.e., $m_k = 0$. This leads to an *omission error* for key k. The second is when at least one record arrives for key k after the final update is flushed; i.e., when $d_{m_k,k} < a_{n_k,k}$. This leads to a *residual error* for key k.

Optimization Metrics

As in Chapter 4, *staleness* is defined as the smallest time interval s such that the results of grouped aggregation for the time window $[T, T+W)$ are available at the center at time $T+W+s$.

Over a window, we define the *per-key error* e_k for a key k to be the difference between the final aggregated value $\hat{V}_k$ and its true aggregated value V_k: $e_k = \text{error}(\hat{V}_k, V_k)$, where error is application-defined.[1] *Error* is then defined as the *maximum* error over all keys:

[1] Our work applies to both absolute and relative error definitions.

$Error = \max_k e_k$. Error arises when an algorithm flushes updates for some keys prior to receiving all arrivals for those keys, or when it omits flushes for some keys altogether.

As we show in the next section, staleness and error are fundamentally opposing requirements. The goal of an aggregation algorithm is therefore either to *minimize staleness given an error constraint*, or to *minimize error given a staleness constraint*.

5.2.2 The Staleness-Error Tradeoff

Mechanics of the Tradeoff

To understand the mechanics behind the staleness-error tradeoff, it is useful to consider what causes staleness and error in the first place. Recall that staleness for a given time window is defined as the delay between the end of that window and the time at which the last update for that window reaches the center.[2] Error is caused by any updates that are not delivered to the center, either because the edge sends only partial aggregates, or because the edge omits some aggregates altogether.

Intuitively, the two main causes of high staleness are either using too much network bandwidth (causing delays due to network congestion), or delaying transmissions until the end of the window. Thus, we can reduce staleness by *avoiding some updates altogether* (to avoid network congestion), and *sending updates earlier during the window*. Unfortunately, both of these options for reducing staleness lead directly to sources of error. First, if no update is ever sent for a given key, then the center never receives any aggregate value for this key, leading to an omission error for the key. Second, if the last update for a key is scheduled prior to the final arrival for that key, then the center will see only a partial aggregate, leading to a residual error for that key. Thus, we see that there is a fundamental tradeoff between achieving low staleness and low error.

As a more concrete example, Figure 5.1 shows the optimal tradeoff between staleness and error for a single window using a `Sum` aggregation over our real-world CDN workload. (Sections 4.3 and 5.3 describe the details of the dataset and algorithms, respectively.) We see that the tradeoff is especially pronounced at low WAN bandwidth, where applications must be error-tolerant in order to maintain bounded staleness.

---

[2] Wide-area *latency* contributes to this delay, but this component of delay is a function of the underlying network infrastructure and is not something we can directly control through our algorithms. We therefore focus our attention on network delays due to wide-area *bandwidth* constraints.

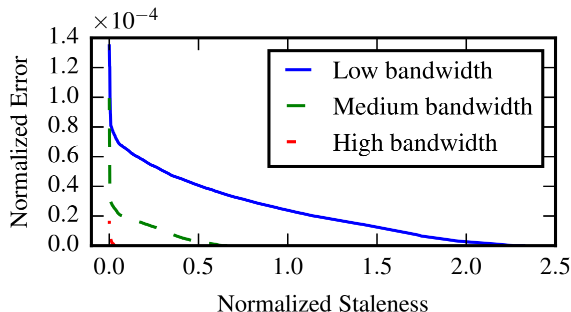


Figure 5.1: The staleness-error tradeoff curve for a `Sum` aggregation over a single time window. Note that we normalize staleness relative to the window length and error relative to the maximum possible error for the window.

Challenges in Optimizing Along the Tradeoff Curve

To understand the challenges of optimizing for either of these metrics while bounding the other, consider two alternate approaches to grouped aggregation: *streaming*, which immediately sends updates to the center without any aggregation at the edge; and *batching*, which aggregates all data during a time window at the edge and only flushes updates to the center at the end of the window. When bandwidth is sufficiently high, streaming can deliver extremely low staleness and high accuracy, as arrivals are flushed to the center without additional delay, and all updates reach the center quickly. When bandwidth is constrained, however, it can lead to both high staleness and error. This is because it fails to take advantage of edge resources to reduce the volume of traffic flowing across the wide-area network, leading to high congestion and unbounded network delays and in turn unbounded staleness.

Batching, on the other hand, leverages computation at edge resources to reduce network bandwidth consumption. When the end of the window arrives, a batching algorithm has the final values for every key on hand, and it can prioritize important updates in order to reach an error constraint quickly, or to reduce error rapidly until reaching a staleness constraint. On the other hand, batching introduces delays by deferring all flushes until the end of the window, leading to high staleness for any given error.

One alternate approach is a *random sampling* algorithm that prioritizes important transmissions after the end of the window, but sends a subset of aggregate values selected randomly during the window. This approach improves over batching by sending updates earlier during the window, thus reducing staleness. It also improves over streaming under bandwidth constraints by reducing the network traffic. As we show in Sections 5.3 and 5.4, however, this approach can still yield high error and staleness due to lack of prioritization among different updates during the window.

To satisfy our optimization goals, a more principled approach is required.

5.3 Offline Algorithms

We now consider the two complementary optimization problems of minimizing staleness (resp., error) under an error (resp., staleness) constraint. Before we present practical online algorithms to solve these problems, we consider optimal *offline* algorithms; i.e., algorithms that have perfect knowledge of all arrivals, even those in the future. Although these offline algorithms cannot be applied directly in practice, they serve both as baselines for evaluating the effectiveness of our online algorithms, and also as design inspiration, helping us to identify heuristics that practical online algorithms might employ in order to emulate optimal algorithms.

5.3.1 Minimizing Staleness (Error-Bound)

The first optimization problem we consider is minimizing staleness under an error constraint (error $\leq E$), where E is an application-specified error tolerance value.

In this case, the goal is to flush only as many updates as is strictly required, and to flush each of these updates as early as possible such that the error constraint is satisfied. Intuitively, an offline optimal algorithm achieves this by flushing an update for each key as soon as the aggregate value for that key falls within the error constraint.

Throughout this section, consider the time window of length W beginning at time T, let $\oplus$ denote our binary aggregation function, and let n denote the number of unique keys arriving during the current window. Define the *prefix aggregate* $V_k(t)$ for key k at time t to be the aggregate value of all arrivals for key k during the current window prior to time t. We define the prefix aggregate $V_k(t)$ to have a logical zero value prior

to the first arrival for key k. Let $\text{error}(\hat{x}, x)$ denote the error of the aggregate value $\hat{x}$ with respect to the true value x. Further, define *prefix error* $e_k(t)$ for key k at time t to be the error of the prefix aggregate for key k with respect to the true final aggregate: $e_k(t) = \text{error}(V_k(t), V_k(T+W))$, for $T \leq t < T+W$. We refer to the prefix error of key k at the beginning of the window—$e_k(T)$—as the *initial prefix error* of key k.

Definition 5.1 (Eager Prefix Error)**.** *Given an error constraint E, the Eager Prefix Error (EPE) algorithm flushes each key k at the first time t such that $e_k(t) \leq E$. If $e_k(T) \leq E$, then EPE avoids flushing key k altogether.*

Theorem 5.1. *The EPE algorithm is optimal.*

Proof. Such an algorithm satisfies the error constraint by construction, and because flushes are issued as early as possible for each key, and only if strictly necessary, there cannot exist another schedule that achieves lower staleness. □

Corollary 5.1. *The EPE algorithm is traffic-optimal for any error bound E.*

The reason is that it flushes only keys with $e_i(T) > E$.

Corollary 5.2. *When $E = 0$, the EPE algorithm achieves the optimal staleness for exact computation.*

This is because, when $E = 0$, this algorithm flushes an update for each key upon the final arrival for that key.[3] It is therefore strictly a generalization of the eager offline optimal algorithm for *exact* windowed grouped aggregation (Section 4.4), which is staleness-optimal.

5.3.2 Minimizing Error (Staleness-Bound)

Next, we consider the optimization problem of minimizing error under a staleness constraint (staleness $\leq S$), where S is an application-specified staleness tolerance (or deadline).

To minimize error under a staleness constraint, we abstract the wide-area network as a sequence of contiguous *slots*, each representing the ability to transmit a single update.

[3] The flush can occur prior to the final arrival if the arrivals for key k end with a sequence of one or more zeros.

The duration of a single slot in seconds is then $\frac{1}{b}$, where b represents the available network bandwidth in updates per second.[4] If S denotes the staleness constraint in seconds, then the final slot ends S seconds after the end of the window. Figure 5.2 illustrates this model for the time window $[T, T+W)$.

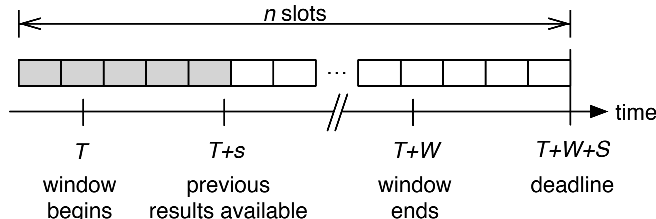


Figure 5.2: We view the network as a sequence of contiguous slots. Shaded slots are unavailable to the current window due to the previous window's staleness.

Note that there is no reason to flush a key more than once during any given window, as any updates prior to the last could simply be aggregated into the final update before it is flushed. Given this fact, we can focus on scheduling flushes for the n unique keys that arrive during the window by assigning each into one of n slots.

Note that flushes from the previous window occupy the network for the first s seconds of the current window, where $s \leq S$ is the staleness of the previous window. These slots are therefore unavailable to the current window, and assigning a key to such a slot has the effect of sending no value for that key. In general, the first slot of the current window may begin prior to this time, or in fact prior to the beginning of the current window.

To illustrate how we assign keys to slots, we first introduce the notion of *potential error*. The potential error[5] $E_k(t)$ for key k at time t is defined to be the error that the center would see for key k if key k were assigned to a slot beginning at time t. Recall that, for $t < T + s$, slots are unavailable, as the network is still in use transmitting updates from the previous window. Assigning a key to such a slot therefore has the effect of sending no value for that key. Similarly, if a key is assigned to a slot prior to that key's first arrival, there is no value to send, so making such an assignment is equivalent to sending no value for that key. We refer to either of these cases as *omitting* a key. Overall then, potential error is given by

[4] In general, bandwidth need not be constant.
[5] Throughout this section, we discuss error with respect to the window $[T, T+W)$.

$$E_k(t) = \begin{cases} e_k(T) & \text{if } t < T + s, \\ e_k(t) & \text{otherwise,} \end{cases}$$

where $e_k(t)$ is the prefix error as defined in Section 5.3.1.

To begin, we assume a *monotonicity property*: the potential error $E_k(t)$ of a key k at any time t is no larger than its potential error $E_k(t')$ at a prior time $t' < t$ within the window, i.e., $E_k(t) \leq E_k(t')$. Intuitively, we assume that the aggregate value for each key only gets *closer* to its true value with additional arrivals. (We relax this assumption shortly.)

Definition 5.2 (Smallest Potential Error First)**.** *The Smallest Potential Error First (SPEF) algorithm iterates over the n slots beginning with the earliest, assigning to each slot a key with the* smallest potential error *that has not yet been assigned to a slot.*

Lemma 5.1. *A schedule D which assigns keys i and j at times t_i and t_j such that $t_i < t_j$ and $E_i(t_i) \leq E_j(t_i)$ (i.e., in SPEF order) cannot have higher error than a schedule D' that swaps these keys.*

Proof. For each key $1 \leq k \leq n$, let t_k be the (only) time at which key k is assigned. The final error is then given by $E = \max_{1 \leq k \leq n} E_k(t_k)$.

Schedule D assigns keys i and j at times t_i and t_j respectively, where $t_i < t_j$, and $E_i(t_i) \leq E_j(t_i)$. The final error E is then bounded by $E \geq \max(E_i(t_i), E_j(t_j))$.

Alternatively, schedule D' swaps these keys, assigning key i at time t_j, and key j and time t_i. This would yield error $E' \geq \max(E_j(t_i), E_i(t_j))$.

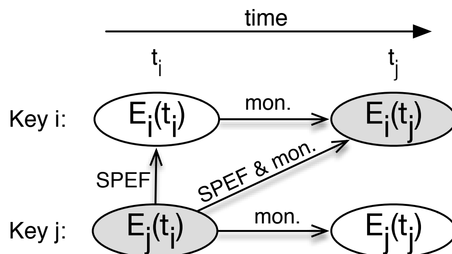


Figure 5.3: Smallest-potential-error-first ordering minimizes error.

Figure 5.3 illustrates the relationships between these different errors. The non-shaded vertices contribute to E, while the shaded vertices contribute to the alternative

error E'. To see why $E' \geq E$, first consider E'. By monotonicity and SPEF ordering, we know that $E_j(t_i) \geq E_i(t_j)$, so it is key j that contributes the greater error to E', and we can describe E' more simply by $E' \geq E_j(t_i)$.

This error cannot be lower than E because, by SPEF ordering, $E_j(t_i)$ is no smaller than $E_i(t_i)$, and by monotonicity, $E_j(t_i)$ is no smaller than $E_j(t_j)$. In other words, whether key i or key j contributes more to the final error in the SPEF-ordered schedule, $E' \geq E$. □

Lemma 5.2. *A schedule D that assigns only $m < n$ keys can be transformed into another schedule D' that assigns n keys without increasing staleness or error.*

Proof. D omits $n - m$ keys. Inserting these $n - m$ keys into the slots left vacant by D yields D'. This transformation cannot increase error because, by monotonicity, sending a key never yields higher error than omitting it does. Further, it cannot increase staleness due to our slot construction. □

Theorem 5.2. *The SPEF algorithm is optimal.*

Proof. The SPEF algorithm satisfies the staleness constraint due to our definition of slots. To see why it minimizes error, let D_{SPEF} denote a sequence of flushes in smallest-potential-error-first order. Let D_{opt} denote an optimal sequence of flushes sharing the longest common prefix with D_{SPEF}. Then there exists an m such that D_{opt} and D_{SPEF} differ for the first time at index m. If D_{SPEF} and D_{opt} are not already identical, then $m < n$ where n is the number of unique keys during the window, and hence the length of D_{SPEF}.

Assume that D_{opt} also has length n. (If not, then transform according to Lemma 5.2.) There must then exist an index $m < p \leq n$ such that $E_m(t_m) \geq E_p(t_m)$. In other words, in slot m, D_{opt} emits a key that does not have the smallest potential error. By Lemma 5.1, we can transform D_{opt} into D'_{opt} by swapping keys m and p without increasing error or staleness. In particular, if $p = \text{argmin}_{m<i\leq n} E_i(t_m)$, then this swap yields D'_{opt} that is optimal and identical to D_{SPEF} up to index $m+1$. This exposes a contradiction: it could not have been true that D_{opt} had the longest prefix in common with D_{SPEF}. Therefore m cannot be less than n: there must exist some optimal departure sequence identical to D_{SPEF}. □

Corollary 5.3. *Let S_{opt} denote the optimal staleness for exact computation. When the staleness constraint $S \geq S_{\text{opt}}$, the SPEF algorithm achieves zero error.*

Intuitively, this is because for an optimal staleness bound S_{opt} (or higher), by definition, the first of the n slots for updates begins late enough that there always exists a key that has attained its final value. The SPEF algorithm then always selects one of these exact keys to be flushed, and hence achieves zero error. At $S = S_{\text{opt}}$, the SPEF algorithm in fact reduces to the lazy optimal algorithm for *exact* computation for an optimal staleness bound (Section 4.4).

An Alternate Optimal Algorithm

Recall that assigning a key to a slot prior to the first arrival for that key, or before the network is available, is equivalent to sending no value for that key; i.e., *omitting* that key. By studying how the SPEF algorithm schedules such omissions, we can derive an alternative optimal algorithm that is more amenable to emulation in an online setting.

Definition 5.3 (SPEF with Early Omissions)**.** *Let m be the number of keys that the SPEF algorithm omits. Then the SPEF with Early Omissions (SPEF-EO) algorithm first omits the m keys with the smallest initial prefix errors, then assigns the remaining $n - m$ keys in SPEF order.*

Lemma 5.3. *A key omitted by the SPEF algorithm has, at the time it is omitted, the smallest initial prefix error among all not-yet-assigned keys.*

Proof. A key is omitted when it is assigned to a slot prior to the first arrival for that key, or before the network is available. In either case, its potential error is equal to its initial prefix error. SPEF assigns keys in smallest-potential-error-first (SPEF) order, so at the time that key i is omitted, its initial prefix error must be smaller than the potential error of all other not-yet-assigned keys. By monotonicity, potential error never exceeds initial prefix error, so key i must also have smaller initial prefix error than all other not-yet-assigned keys. □

Note that this is a necessary but insufficient condition for a key to be omitted: another key may have a smaller potential error than the smallest initial prefix error.

Theorem 5.3. *The SPEF-EO algorithm is optimal.*

Proof. By Lemma 5.3, the final key omitted by the SPEF algorithm contributes the greatest error of all prior omissions, and this error is no less than the mth-smallest initial prefix error. Error therefore cannot be reduced by flushing (i.e., not omitting) any of the m keys with the smallest initial prefix errors.

The remaining $n - m$ slots occur no earlier than the $n - m$ slots carrying non-zero updates in the original algorithm. By monotonicity, assigning the remaining $n - m$ keys to these slots therefore cannot increase error. □

Relaxing the Monotonicity Assumption

Up to this point, we have assumed that prefix error decreases monotonically in time, but this assumption can be relaxed in a straightforward manner. Consider an aggregation for which prefix error is non-monotonic. Let the *minimum potential error* for key i at time t be the smallest potential error achieved up until time t during the current window. This value decreases monotonically whether or not the underlying error is monotonic. Therefore the following must hold.

Corollary 5.4. *If flushed updates always contain the value that achieves the minimum potential error, then assigning to each slot the key with the smallest* minimum potential error *is an optimal algorithm.*

5.3.3 High-Level Lessons

These offline optimal algorithms, although not applicable in practical online settings, leave us with several high-level lessons. First, they flush each key at most once, thereby avoiding wasting scarce network resources. Second, they make the best possible use of network resources. In the error-bound case, EPE achieves this by sending only keys that have already satisfied the error constraint. For the staleness-bound case, with SPEF and SPEF-EO, this means using each unit of network capacity (i.e., each slot) to send the most up-to-date value for the key with the minimum potential error, and for SPEF-EO, sending only those keys with the largest initial prefix errors.

5.4 Online Algorithms

We are now prepared to consider practical online algorithms for achieving near-optimal staleness-error tradeoffs. The offline optimal algorithms from Section 5.3 serve as useful baselines for comparison, and they also provide models to emulate.

To evaluate alternative design decisions and to compare our proposed algorithms to a number of baseline algorithms, we reuse the simulation methodology introduced in Section 4.5. Simulation allows us to more rapidly explore a large design space, to compare against an offline optimal algorithm, and to select a practical baseline approach to use in our experiments later.

Throughout this chapter, simulation results present median values (staleness or error) over 25 consecutive time windows from the Akamai trace, spanning multiple days' worth of workload. We normalize error by the largest possible error for our trace, and we normalize staleness by the window length.

5.4.1 The Two-Part Cache Abstraction

Exact Computation

Our online algorithms for approximate windowed grouped aggregation generalize those for exact computation (Chapter 4), where we view the edge as a *cache* of aggregates and emulate offline optimal algorithms through cache *sizing* and *eviction* policies. When a record arrives at the edge, it is inserted into the cache, and its value is merged via aggregation with any existing aggregate sharing the same key. The sizing policy, by dictating how many aggregates may reside in the cache, determines *when* aggregates are evicted. For exact computation, an ideal sizing policy allows aggregates to remain in the cache until they have reached their final value, while avoiding holding them so long as to lead to high staleness. The eviction policy determines *which* key to evict, and an ideal policy for exact computation selects keys with no future arrivals. Upon eviction, keys are enqueued to be transmitted over the WAN, which we assume services this queue in FIFO order.

Approximate Computation

For approximate windowed grouped aggregation, we continue to view the edge as a cache, but we now partition it into *primary* and *secondary* caches, as illustrated in Figure 5.4.

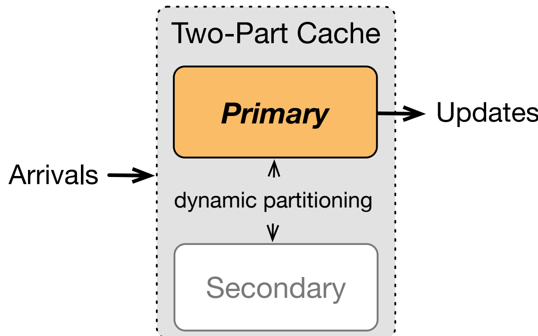


Figure 5.4: The cache is dynamically partitioned into primary and secondary parts. Updates are flushed only from the primary cache.

The reason for this distinction is that, when approximation is allowed, it is no longer necessary to flush updates for all keys. An online algorithm must determine not just *when* to flush each key, but also *which* keys to flush. The distinction between the two caches serves to answer the latter question: *updates are flushed only from the primary cache*. It is the role of the cache *partitioning* policy to define the boundary between the primary and secondary cache, and the main difference between our error-bound and staleness-bound online algorithms lies in the partitioning policy. As we discuss throughout this section, our error-bound algorithm defines this boundary based on the *values* of items in the cache, while the staleness-bound algorithm uses a dynamic *sizing* policy to determine the size of the primary cache.

In addition, the primary cache logically serves as the outgoing network queue: updates are flushed from this cache when network capacity is available. Unlike FIFO queuing, this ensures that our online algorithms make the most effective possible use of network resources. In particular, it ensures that flushed updates always reflect the most up-to-date aggregate value for each key. Additionally, it allows us to use our eviction policies to choose the most valuable key to occupy the network at any time.

5.4.2 Error-Bound Algorithms

Cache Partitioning Policy

Our online error-bound algorithm uses a value-based cache partitioning policy, which defines the boundary between primary and secondary caches in terms of aggregate values. It is inspired by the offline optimal EPE algorithm (Section 5.3.1) that only flushes keys whose prefix error is within the error bound. Specifically, new arrivals are first added to the secondary cache, and they are *promoted* into the primary cache only when their aggregate grows to exceed the error constraint. More rigorously, let F_k denote the total aggregate value flushed for key k so far during the current window (logically zero if no update has been flushed), and let V_k denote the aggregate value currently maintained in the cache for key k. Then the *accumulated error* for key k is defined as $\mathrm{error}(F_k, F_k \oplus V_k)$; i.e., the error between the value that the center currently knows and the value it would see if key k were flushed.[6] Key k is moved from the secondary cache to the primary cache when its accumulated error exceeds E. Given this policy, and the fact that updates are only flushed from the primary cache, we are guaranteed to flush only keys that strictly must be flushed in order to satisfy the error constraint; i.e., this approach avoids false positives.

On the other hand, this approach only *eventually* avoids false negatives. That is, a key that will eventually need to be flushed may remain in the secondary cache beyond the time at which its prefix error falls below the error bound E, since the final value for a key is unknown in an online setting. A more *eager* variant would promote a key into the primary cache when it is probable—though not necessarily certain—that it will require flushing. We do not explore such alternatives in depth here, however, as our simulation results have shown that these transient false negatives contribute only a small amount to staleness.

After the end of the window, our online algorithm flushes the entire contents of the primary cache, as all of its constituent keys exhibit sufficient accumulated error to violate the error constraint if not flushed.[7]

[6] Note we can compute this value online.

[7] The order of these flushes is unimportant, as all of them are required to bring error below E.

Cache Eviction Policy

Prior to the end of the window, some prediction is involved: When the network pulls an update from the primary cache, the eviction policy must identify a key that has reached an aggregate value within E of its final value. We explore three practical cache replacement policies: least-recently used (LRU), smallest potential error (SPE), and most accumulated error (MAE).

Intuitively, LRU assumes that the key with the least recent arrival is also the least likely to receive future arrivals, and hence closest to its final value. Therefore LRU evicts the key with the least recent arrival.

SPE tries to faithfully emulate the offline optimal algorithm by explicitly predicting the final value for each key, and it uses this prediction to estimate the potential error of each key, ultimately evicting the key with the smallest estimated potential error.

The MAE policy is greedy in nature: it evicts the key that will make the largest reduction in error; i.e., the key with the largest accumulated error currently.

Figure 5.5 compares these simple and practical eviction policies for a `Sum` aggregation. We find that LRU and SPE perform about equally and both slightly outperform MAE, especially at lower error limits. LRU may be preferable to SPE in practice because of its simplicity.

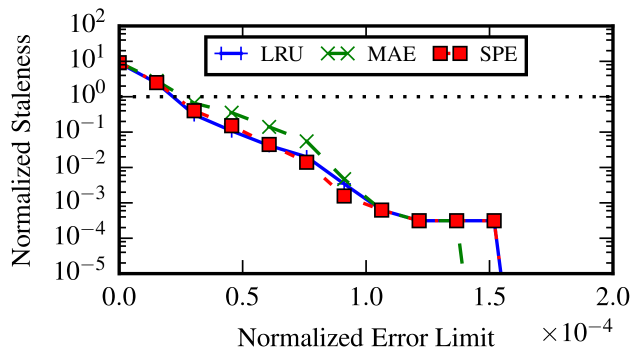


Figure 5.5: Impact of primary cache eviction policy on staleness given an error bound. Note the logarithmic y-axis scale.

The main point of these results is that it is not necessary to develop a sophisticated

prediction policy to yield low staleness under an error constraint. Instead, *even simple and practical policies are very effective.*

5.4.3 Staleness-Bound Algorithms

An online algorithm that aims to minimize error under a staleness constraint faces slightly different challenges. In particular, we can no longer define the boundary between primary and secondary caches by value, as we do not know a priori what this value should be. Instead our staleness-bound online algorithm uses a dynamic *sizing-based* cache partitioning policy to emulate the offline optimal SPEF-EO algorithm (Section 5.3.2). To understand this approach, recall from Definition 5.3 that the SPEF-EO algorithm flushes only the keys with the largest initial prefix errors. Given that the role of the primary cache is to contain the keys that must be flushed, we emulate this behavior by 1) dynamically ranking cached keys by their (estimated) initial prefix error, and then 2) defining the primary cache at time t to comprise the top $\sigma(t)$ keys in this order. In practice, this is challenging as we must predict both initial prefix errors as well as an appropriate sizing function $\sigma(t)$.

Initial Prefix Error Prediction

During the window, there are many ways to predict initial prefix errors. One straightforward approach is to use accumulated error (Section 5.4.2) as a proxy for initial prefix error. We call this the *Acc* policy for short. Intuitively, this policy assumes that the keys that have accumulated the most error so far also have the largest initial prefix errors, and therefore ranking keys by accumulated error is a good approximation for ranking them by initial prefix error. An alternative approach—which we simply call *Explicit*—attempts to explicitly predict initial prefix errors by predicting the final value for each key.

Figure 5.6 compares these approaches for `Sum` aggregation, and shows that the very simple Acc policy typically yields lower error than a more complex approach does. This again demonstrates that sophisticated prediction approaches are not required in order to yield good performance with our two-part caching approach; even very simple and practical policies can be effective.

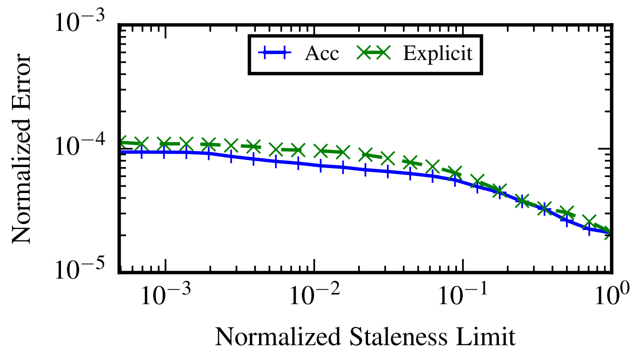


Figure 5.6: Impact of the initial prefix error prediction algorithm on staleness given error bound. Note the logarithmic axis scales.

Cache Size Prediction

Predicting the appropriate primary cache size function $\sigma(t)$ raises several additional challenges. To begin, consider how to define this function in an ideal world where we have a perfect eviction policy and a perfect prediction of initial prefix error. At time t, the primary cache size $\sigma(t)$ represents the number of keys that are *present* in the primary cache. These $\sigma(t)$ keys will need to be flushed prior to the staleness bound, as will any not-yet-arrived keys with sufficiently high initial prefix error. Let $f(t)$ denote the number of future arrivals of these large-prefix-error keys. Then, if the total number of network slots remaining until the staleness constraint is given by $B(t)$, an ideal primary cache size function $\sigma(t)$ satisfies $B(t) = \sigma(t) + f(t)$.

In practice, we have neither a perfect eviction policy, nor a perfect prediction of initial prefix error, and determining $f(t)$ is a nontrivial prediction on its own. We have explored a host of alternative approaches, and we highlight three here: *PrevWindow*, *Linear*, and *PurePess*.

The PrevWindow policy assumes that arrivals in the current window will resemble those in the previous window, and it therefore uses the previous window's arrival sequence to compute $f(t)$. Using an exponentially weighted moving average of the available WAN bandwidth to predict $B(t)$, this $f(t)$ is then used to compute the appropriate primary cache size function $\sigma(t)$.

The Linear policy makes the simplifying assumption that keys with the largest initial prefix errors have initial arrivals distributed uniformly throughout the window. In other words, if we assume k such keys will arrive during the window, then for the time window $[T, T+W)$, we assume $f(t)$ decays linearly as $f(t) = \frac{k(T-t)}{W}$. We estimate k based on the predicted number of slots available for the window; i.e., $k = B(T)$.

The PurePess policy makes a pessimistic assumption about future arrivals, specifically that all future arrivals are for keys that should occupy the primary cache. We monitor the arrival rate of new keys using an exponentially weighted moving average, and base $f(t)$ on an extrapolation of this average rate. As a result, when bandwidth is highly constrained, the primary cache is effectively nonexistent, and this policy degrades to evicting keys in order of estimated initial prefix error.

Figure 5.7 compares these policies for `Sum` aggregation, and shows that the PrevWindow policy nearly universally outperforms the others. PurePess also performs reasonably well, but is slightly worse than PrevWindow. The Linear policy performs poorly at all but the highest staleness constraints.

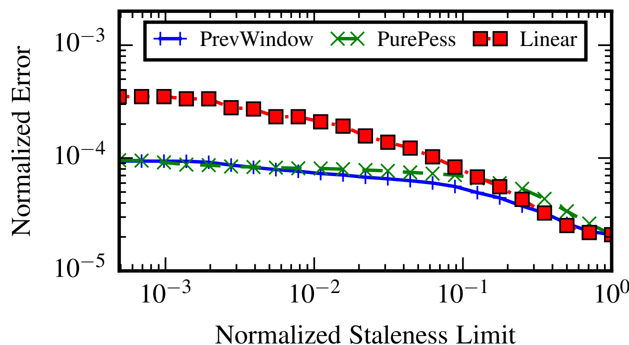


Figure 5.7: The impact of primary cache sizing policy on error given staleness bound. Note the logarithmic axis scales.

Cache Eviction Policy

During the window, we use the sizing-based cache partitioning policy to delineate the boundary between primary and secondary caches. When network capacity is available, it

remains the role of the cache eviction policy to determine which key should be evicted from the primary cache. As in Section 5.4.2, we again find that simple well known policies such as LRU perform well.

Upon reaching the end of the window, there is no need for prediction since all final aggregate values are known. At this point, keys are evicted from the union of the primary and secondary caches in descending order by their accumulated error until reaching the staleness bound.

5.4.4 Implementation

We implement our algorithms in Apache Storm [58], extending our prototype from Chapter 4 for exact computation. Here we highlight the changes from our earlier prototype.

At the edge, the `Replayer` spout is unaltered, while the `Aggregator` bolt is extended to implement our two-part cache. This two-part cache is implemented using efficient priority queues, allowing us to generalize over a range of cache partitioning and eviction policies in order to implement our online algorithms. At the center, `Aggregator` bolts are extended to emit additional summary metrics describing error.

Our online algorithms treat the two-part cache as the outgoing network queue, but Storm does not implement flow control at the bolt level, and the Storm programming model only allows bolts to *push* tuples downstream. To resolve this apparent mismatch, we track queuing delays within both the `SocketSender` and the `SocketReceiver`. These delays are communicated upstream, and the edge `Aggregator`s use them as congestion signals, applying an additive-increase / multiplicative-decrease (AIMD) approach to dynamically adjust the rate at which they push records downstream.

5.5 Evaluation

In this section, we evaluate our online algorithms using both a trace-driven simulation (Section 5.5.3) as well as experiments using our Storm-based implementation on PlanetLab [84] and local testbeds (Section 5.5.5).

5.5.1 Setup

Throughout our experiments, we use two combinations of PlanetLab nodes at the University of Oregon as well as local nodes at the University of Minnesota. The Oregon site includes three nodes, each with 4 GB of physical memory and four CPU cores. The Minnesota site includes up to eight nodes, each with 12 GB of physical memory and eight CPU cores. WAN bandwidth from Minnesota to Oregon averages 12.5 Mbps based on `iperf3` measurements.

Throughout our evaluation, we use the anonymized Akamai trace (Section 4.3) as input, and we consider several diverse aggregations: `Sum`, `Max`, `Count`, and `HyperLogLog` [65]. Each of these aggregations uses the `large` query key. The `Sum` aggregation computes the total number of bytes successfully downloaded for each key, while the `Max` aggregation computes the largest successful download size for each key. The `Count` aggregation counts the number of arrivals for each key, and the `HyperLogLog` aggregation is used to approximate the number of unique client IP addresses for each key. This demonstrates an important point: our techniques apply to a broad range of aggregations, from distributive and algebraic [66] aggregates such as `Sum`, `Max`, or `Average` to holistic aggregates such as unique count via `HyperLogLog`.[8]

To explore how our algorithms perform under varied WAN bandwidth, we consider *low*, *medium*, and *high* bandwidths, the details of which differ slightly between simulations and experiments. In our simulations, medium and high bandwidth denote three and ten times higher than low bandwidth respectively. In experiments, low and high bandwidth denote 60% and 160% of the medium bandwidth, respectively.

5.5.2 Comparison Algorithms

We compare the following aggregation algorithms.

Baselines

Streaming: A streaming algorithm performs no aggregation at the edge, and instead simply flushes each key immediately upon arrival; that is, it streams arrivals directly

[8] We adopt the common approach [85] of approximating holistic aggregates (e.g., unique count, Top-K, heavy hitters) using sketch data structures (e.g., HyperLogLog, CountMinSketch [86])

on to the center. An error-bound variant continues streaming arrivals until the error constraint E is eventually satisfied, while a staleness-bound variant continues streaming until reaching the staleness limit S.

Batching: A batching algorithm aggregates arrivals until the end of the window without sending any updates. At the end of the window at time $T + W$, an error-bound variant flushes all keys for which omitting an update would exceed the error bound; i.e., all keys with initial prefix error greater than E. A staleness-bound variant begins flushing keys at the end of the window, and does so in descending order of initial prefix error, stopping upon reaching the staleness limit S.

Batching with Random Early Updates (Random): The Random algorithm effectively combines batching with streaming using random sampling. Concretely, the Random algorithm aggregates arrivals at the edge as batching does, but it sends updates for random keys during the window whenever network capacity is available. When the end of the window arrives, it flushes keys as batching does, in decreasing order by their impact on error. This algorithm is a useful baseline to show that the choice of which key to flush during the window is critical.

Offline Optimal

To provide an optimal baseline for evaluating our online algorithms, we implement the EPE and SPEF-EO algorithms from Section 5.3 for the error-bound and staleness-bound optimizations, respectively. Note that these are offline algorithms and are therefore confined to simulation.

Online

These refer to our caching-based online algorithms presented in Section 5.4. For the error-bound case, we use the emulated EPE algorithm, consisting of the two-part caching approach with an LRU primary cache eviction policy, as discussed in Section 5.4.2. For the staleness-bound case, we employ the emulated SPEF-EO algorithm discussed in Section 5.4.3, consisting of LRU eviction, Accumulated Error-based estimation, and PrevWindow cache sizing policies.

5.5.3 Simulation Results: Minimizing Staleness (Error-Bound)

Before presenting experimental results on our PlanetLab deployment, we use the simulation framework introduced in Section 4.5.1 to compare our online algorithms to several baseline algorithms.

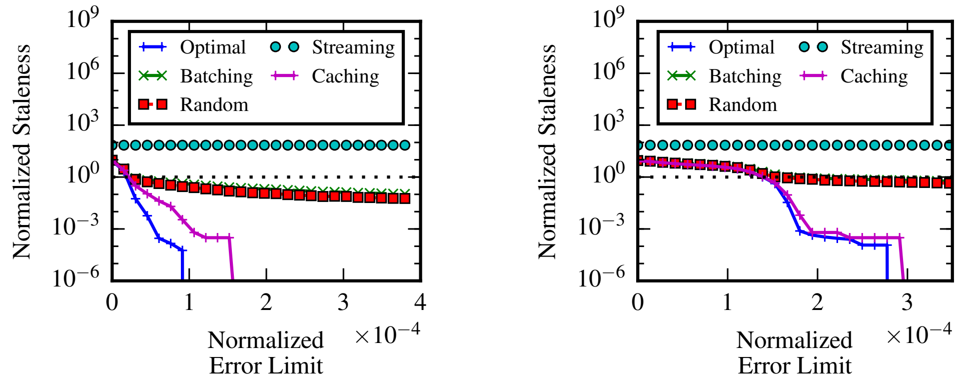
We first compare our caching-based online algorithm against the baseline algorithms for error-constrained applications, where the objective is to minimize staleness for a given error constraint.

Various Aggregations

Figure 5.8 shows staleness (normalized relative to the window length) for a range of error bounds E (normalized relative to the largest possible error for our trace) for our online algorithm, as well as for our three baseline algorithms and an optimal offline algorithm. It shows the comparison for three different aggregation functions under low bandwidth: `Sum`, `Max`, and `Count`.

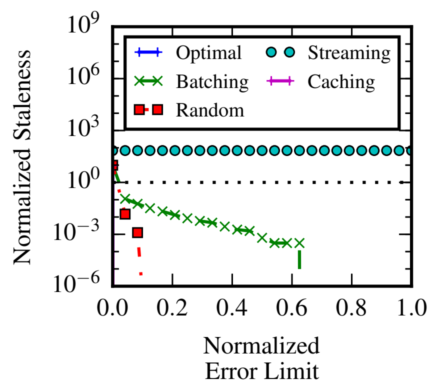
For all aggregations, it is clear that streaming is infeasible, as staleness far exceeds the window length (shown as the dotted horizontal line). By performing aggregation at the edge, batching significantly improves upon streaming, but it still yields high staleness because it delays all flushes until the end of the window. Random improves over batching, but the improvement is minor for `Sum` and `Max`, illustrating the limitation of our random sampling approach. Our caching-based online algorithm, on the other hand, chooses which keys to flush during the window in a principled manner, and yields staleness much closer to an offline optimal algorithm as a result.

It is interesting to contrast `Sum` and `Max` aggregations with the `Count` aggregation. For `Count`, we see that Batching performs reasonably well, while Random offers a significant improvement. Both the offline optimal and our caching algorithm achieve near-zero staleness across a broad range of error constraints. The reason for the difference is that, with a `Count` aggregation, every arrival has the same impact on error (i.e., an increment to the count) whereas for `Sum` and `Max`, arrivals can have an arbitrarily large impact. For `Count`, therefore, it is somewhat easier to predict which keys are closer to their final values and as a result are safe to flush.



(a) `Sum` aggregation.

(b) `Max` aggregation.

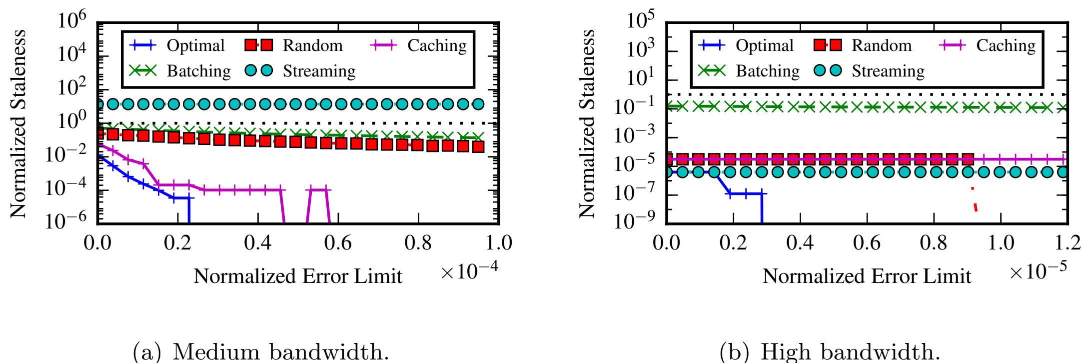


(c) `Count` aggregation.

Figure 5.8: Normalized staleness for various aggregations and error bounds. Note the logarithmic y-axis scales.

Various Bandwidth Constraints

We next examine the impact of WAN bandwidth constraints. Specifically, we run our algorithms under medium and high bandwidth (3 and 10 times higher bandwidth than the low bandwidth used for Figure 5.8, respectively). Figure 5.9 shows normalized staleness for a range of error bounds for `Sum` aggregation under medium and high bandwidth.



(a) Medium bandwidth.

(b) High bandwidth.

Figure 5.9: Normalized staleness for `Sum` aggregation with various error bounds under medium and high bandwidth. Note the logarithmic y-axis scales.

As we might expect, we see that staleness for any given error constraint decreases as bandwidth increases, and this trend holds for all algorithms. At medium bandwidth (Figure 5.9(a)), our caching algorithm continues to outperform the baseline algorithms, and approaches the performance of an offline optimal algorithm. When bandwidth is very high (Figure 5.9(b)), we see that streaming performs very well, as it flushes updates without delay, and incurs no network queuing delays in the process, as described in Section 5.2.2. In this case, our online caching-based algorithm also achieves very low staleness, demonstrating its applicability across a broad range of WAN bandwidths.

5.5.4 Simulation Results: Minimizing Error (Staleness-Bound)

Various Aggregations

Figure 5.10 demonstrates the effectiveness of our approach compared to the other baselines for the case where we are minimizing the error given a staleness limit. The figure shows the normalized error values obtained for different staleness limits. We again see

that streaming is impractical, as its fails to effectively use the limited WAN capacity to transmit the most important updates. Batching yields a significant improvement over streaming, as long as the tolerance for staleness is sufficiently high. Batching with random early updates represents a further improvement, though again, except for `Count` aggregation, the benefit of these early updates is not dramatic. Our caching-based online algorithm, on the other hand, by making principled decisions about which keys to flush during the window, yields near-optimal error across a broad range of staleness limits.

Note that all algorithms except streaming converge to the same error as the staleness bound reaches one window length. The reason is that, as the staleness bound approaches the window length, there are fewer and fewer opportunities to flush updates during the window, and these algorithms converge toward pure batching.

Various Bandwidth Constraints

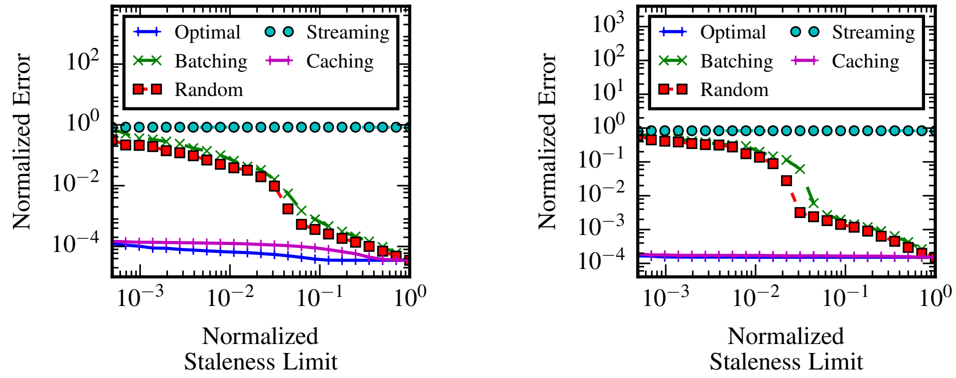
Figure 5.11 shows the same algorithms applied at medium and high bandwidth for the `Sum` aggregation, where the same general observations hold. Echoing the results for our error-bound algorithms in Figure 5.9, Figure 5.11 shows that our caching-based algorithm outperforms the other baselines whether or not bandwidth is constrained, and all algorithms except Batching perform well under very high bandwidth.

5.5.5 Experimental Results

To demonstrate how our techniques perform in a real-world setting, we now present results from an experimental evaluation conducted on PlanetLab [84] using our Storm-based implementation.

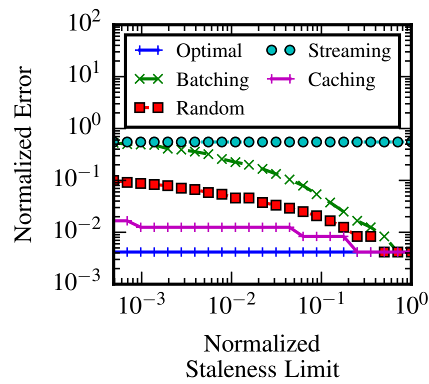
Staleness and error are measured at the center, and we treat the staleness constraint as a hard deadline: when the deadline is reached, we compute the error based on records that have arrived at the center up to that point. In general, there may be arrivals after the deadline due to varying WAN delays, but we disregard these late arrivals as they violate the staleness constraint.

Based on our results from Subsection 5.5.3, we use Random as a practical baseline algorithm for comparison, since it outperforms both batching and streaming.



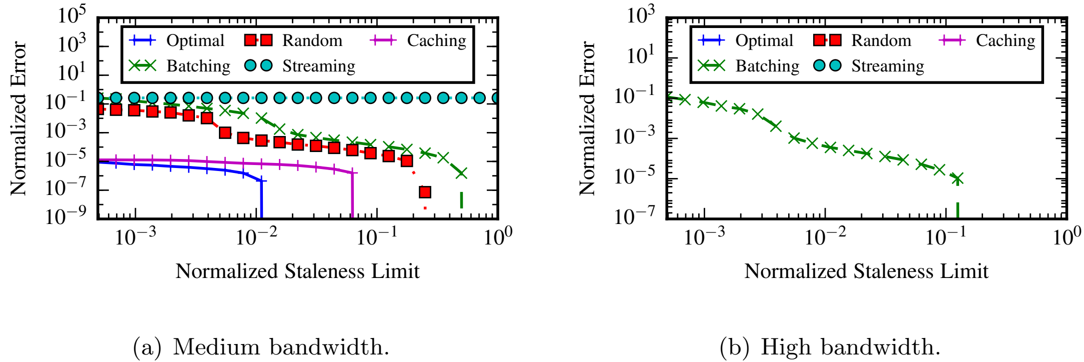
(a) `Sum` aggregation.

(b) `Max` aggregation.



(c) `Count` aggregation.

Figure 5.10: Normalized error for various aggregations and staleness bounds. Note the logarithmic axis scales.



(a) Medium bandwidth.

(b) High bandwidth.

Figure 5.11: Normalized error for `Sum` aggregation with various staleness bounds under medium and high bandwidth. Note the logarithmic y-axis scale.

Geo-Distributed Aggregation

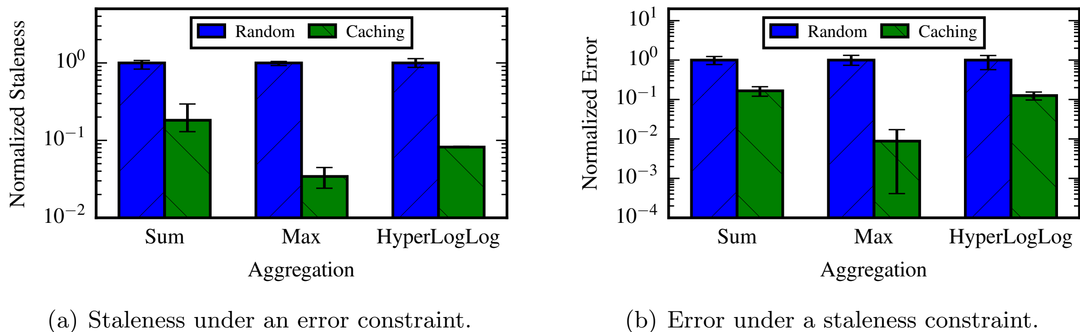
In order to demonstrate our techniques in a real geo-distributed setting, we begin with experiments using Minnesota nodes as the edge and Oregon nodes as the center. Note that we use local Minnesota nodes rather than PlanetLab nodes as the edge because PlanetLab imposes restrictive daily limits on the amount of data sent from each node, rendering repeated experiments with PlanetLab edge nodes impractical.[9]

Figure 5.12(a) and Figure 5.12(b) show staleness and error for error-constrained and staleness-constrained algorithms, respectively. Both staleness and error are normalized relative to the Random baseline for ease of interpretation. We consider the median staleness or error during a relatively stable portion of the workload, and plot the mean over five runs. Error bars show 95% confidence intervals [87].

Figure 5.12 demonstrates that our Caching algorithms significantly outperform a practical random sampling/batching-based baseline in a real-world geo-distributed deployment. Staleness is reduced by 81.8% to 96.6% under an error constraint, while error is reduced by 83.4% to 99.1% under a staleness constraint. Note that the reduction is largest for `Max` aggregation, as many keys attain their final value well in advance of the end of the window, allowing greater opportunity for early evictions.

These results demonstrate that our techniques perform well in a real geo-distributed deployment over real data, and they do so for a diverse set of aggregations.

[9] For example, each one of the five repetitions of the `HyperLogLog` experiments was enough to exhaust the daily bandwidth limit.



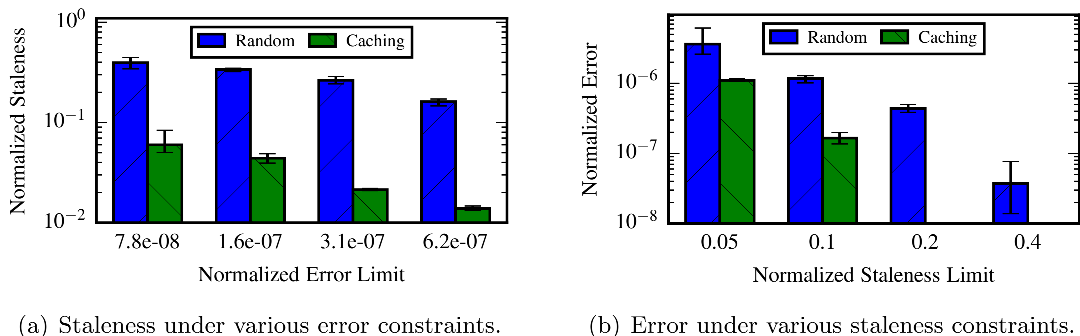
(a) Staleness under an error constraint.

(b) Error under a staleness constraint.

Figure 5.12: Comparison of Random and Caching algorithms for various aggregation functions on a PlanetLab testbed. Note the logarithmic y-axis scale.

Various Error and Staleness Constraints

We further study the performance of our algorithms under various error and staleness constraints. Due to PlanetLab bandwidth limitations, we use an alternative deployment with eight total Minnesota nodes: four as the edge and four as the center, using Linux traffic control (`tc`) to emulate constrained WAN bandwidth between the edge and center clusters. Figure 5.13 shows normalized staleness and error for the `Sum` aggregation. Likewise, Figure 5.14 shows normalized staleness and error for the `Max` aggregation.

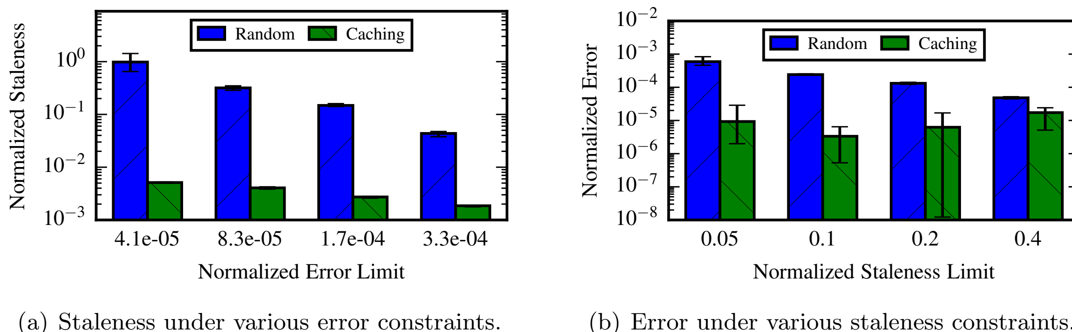


(a) Staleness under various error constraints.

(b) Error under various staleness constraints.

Figure 5.13: Comparison of Random and Caching algorithms for `Sum` aggregation with various error and staleness constraints on a local testbed. Note the logarithmic y-axis scale.

We observe that our Caching algorithms outperform the Random baseline across all error and staleness constraints used in our experiments. For the most part, as the



(a) Staleness under various error constraints.

(b) Error under various staleness constraints.

Figure 5.14: Comparison of Random and Caching algorithms for `Max` aggregation with various error and staleness constraints on a local testbed. Note the logarithmic y-axis scale.

error (respectively, staleness) limit increases, the staleness (respectively, error) decreases, demonstrating the fundamental staleness-error tradeoff that underlies our algorithm development.

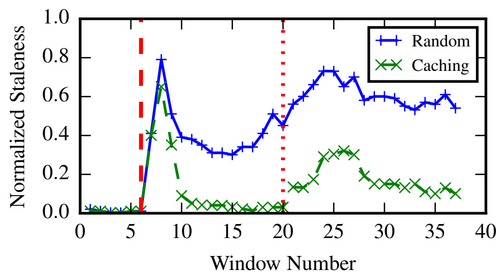
One interesting divergence from this pattern is the `Max` aggregation under a staleness constraint, where Figure 5.14(b) shows that, although our Caching algorithm significantly outperforms the Random baseline, it does not show a significant decrease in error as the staleness limit increases. One possible explanation is that, since many keys reach their final value well in advance of the end of the window, our Caching algorithm performs well even when the staleness limit is low and it is forced to make many early evictions. Further study of this behavior may help us develop more effective partitioning and eviction policies.

Dynamic Workload and WAN Bandwidth

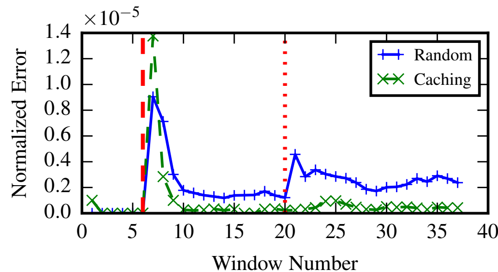
To dig deeper, we assess how our algorithms perform under dynamic workload and WAN bandwidth. We use `tc` to control the emulated WAN bandwidth between the edge and center clusters. By changing the emulated WAN bandwidth at runtime, we can study how our algorithms adapt to dynamic WAN bandwidth. Further, our anonymized Akamai trace exhibits a significant shift in workload early in the trace, providing a natural opportunity to study performance under workload variations.

Figure 5.15 shows how our staleness-bound and error-bound algorithms behave under

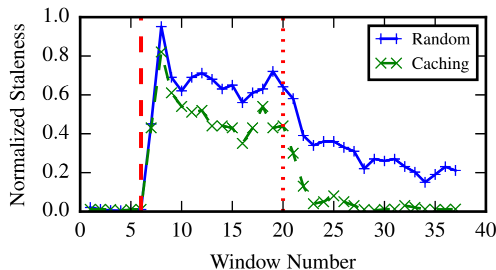
these workload and bandwidth changes for the `Sum` aggregation. To illustrate the effect of a bandwidth decrease, Figure 5.15(a) plots a time series of staleness normalized relative to the window length, while Figure 5.15(b) plots error normalized relative to the largest possible error. Similarly, Figure 5.15(c) and Figure 5.15(d) plot error and staleness, respectively, under increasing bandwidth. For each of these figures, we run the experiment five times, and plot the median staleness or error for each window.



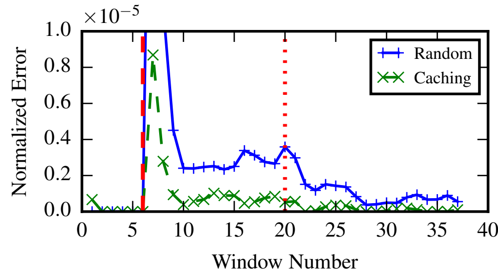
(a) Bandwidth decrease: Staleness under an error constraint.



(b) Bandwidth decrease: Error under a staleness constraint.



(c) Bandwidth increase: Staleness under an error constraint.



(d) Bandwidth increase: Error under a staleness constraint.

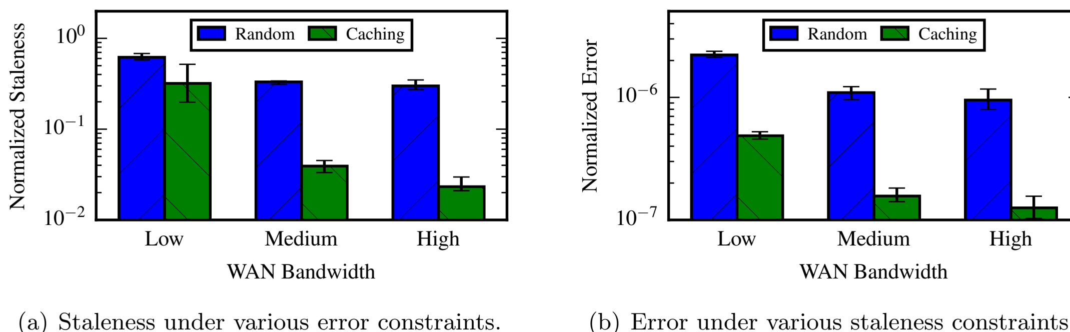
Figure 5.15: Comparison of Random and Caching algorithms for `Sum` aggregation under dynamic workload and WAN bandwidth on a local testbed. Arrival rate increases during window 6 (dashed vertical line), and available bandwidth changes during window 20 (dotted vertical line).

We see that both Random and Caching algorithms require some time to adapt to the workload change that occurs during window 6. Once stabilized, Caching yields significantly lower staleness and error than Random. During window 20, we change the (emulated) WAN bandwidth. In Figures 5.15(a) and 5.15(b), bandwidth is reduced by 62.5%, and while staleness and error increase for both algorithms, our Caching

algorithms continue to significantly outperform the Random baseline. In Figures 5.15(c) and 5.15(d), bandwidth increases by 167% during window 20, and we again observe that our Caching algorithms adapt to the change and continue to outperform the Random baseline.

Various Bandwidth Constraints

To better understand how our algorithms perform under varied WAN bandwidth, we emulate low, medium and high bandwidths using `tc` on our eight-node cluster. Figure 5.16(a) shows normalized staleness for a fixed moderate error constraint for a `Sum` aggregation under the three bandwidth constraints. We see that our Caching algorithm yields significantly lower staleness than the Random baseline. Similarly, Figure 5.16(b) shows normalized error under a fixed moderate staleness constraint. We observe that our Caching algorithm significantly reduces error relative to the Random baseline. From a higher level, Figure 5.16 shows that, for both error-bound and staleness-bound applications, lower bandwidth leads to higher staleness (resp., error) for a given error (resp., staleness) constraint, just as we saw in Section 5.5.3.



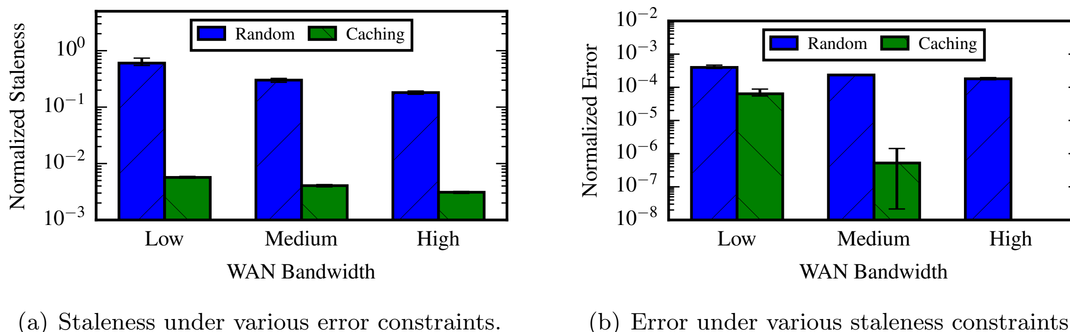
(a) Staleness under various error constraints.

(b) Error under various staleness constraints.

Figure 5.16: Comparison of Random and Caching algorithms for `Sum` aggregation with various emulated WAN bandwidths on a local testbed. Note the logarithmic y-axis scale.

Figure 5.17 repeats these experiments for a `Max` aggregation. We again see the increasing magnitude of the staleness-error tradeoff as available bandwidth decreases.

In addition, we see that at high bandwidth, we can achieve zero error for the `Max` aggregation while `Sum` showed small but nonzero error. The reason is that, for `Max`



(a) Staleness under various error constraints.

(b) Error under various staleness constraints.

Figure 5.17: Comparison of Random and Caching algorithms for `Max` aggregation with various emulated WAN bandwidths on a local testbed. Note the logarithmic y-axis scale.

aggregation, many keys take on their final aggregate value well before the window ends. (When the maximum value for a key has already been observed, no additional arrivals will change the maximum for that key.) For `Sum` on the other hand, every (nonzero) arrival is guaranteed to affect the value for its key. As a result, flushing keys during the window carries less risk for `Max`. Relatively few keys see large enough arrivals after their initial flush in order to require a second flush. In practice, this means that the highest bandwidth in our experiments is sufficient to communicate exact values for every key when using a `Max` aggregation, but is not quite enough to accommodate the additional burden of re-flushing keys that were flushed too early with the `Sum` aggregation.

Relative Impact of Partitioning and Eviction Policies

In order to understand the relative importance of the cache eviction and cache partitioning policies, we run additional experiments for our error-constrained algorithm. We compare the Random and Caching algorithms along with a third alternative, the *Random (2-part)* algorithm, where we use our two-part Caching approach with a random eviction policy. This third alternative allows us to separate the effects of cache partitioning and cache eviction policies.[10]

Figure 5.18 compares these policies for the `Sum` and `Max` aggregation at medium WAN bandwidth. For the `Sum` aggregation, we see that Random (2-part) does not

[10] For the staleness-constrained case, separating these is not straightforward, so we evaluate only the error-constrained case.

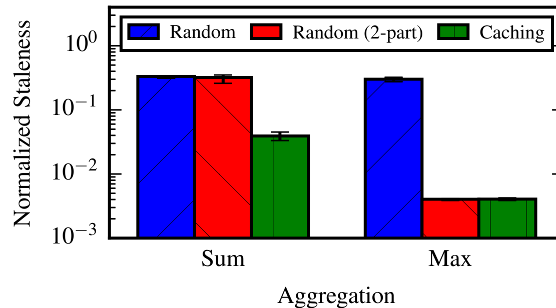


Figure 5.18: Comparison of Random, Random (2-part), and Caching algorithms for `Sum` and `Max` aggregations at medium WAN bandwidth. Note the logarithmic y-axis scale.

yield significantly lower staleness than the baseline Random algorithm, but our Caching algorithm does. This demonstrates that the partitioning policy alone is not effective in reducing staleness, but the combination of partitioning and eviction policies is. The reason is that, throughout much of the window, many keys have accumulated error larger than the error limit, yet many of these keys will still receive significant future updates and should therefore not yet be flushed. The eviction policy serves to identify keys that are more likely to be safe for eviction.

For the `Max` aggregation, on the other hand, both Random (2-part) and our Caching algorithm significantly outperform the Random baseline, suggesting that the partitioning policy plays the major role in reducing staleness. The reason is that, as the window advances, fewer and fewer keys have accumulated error larger than the error limit, so the partitioning policy alone is quite effective in identifying important keys to flush. Further, there is a weak correlation between recency of arrivals and the extent to which the maximum value will continue to increase for a given key, hence an LRU eviction policy is not significantly more effective than a random policy for this aggregation.

Overall, our experiments demonstrate that our Caching algorithms apply across a diverse set of aggregations, and that they outperform the Random baseline in real geo-distributed deployments, under dynamic workloads and bandwidth conditions, and across a range of error and staleness constraints.

5.6 Discussion

5.6.1 Alternative Approximation Approaches

In this chapter, we have considered one particular approach to approximation: flushing some keys early while omitting some others altogether. Essentially, this approach samples a prefix for each key, and applies the aggregation function directly to this prefix.

A major advantage of this approach is its simplicity; we can apply this prefix-sampling approach to any windowed grouped aggregation. One disadvantage, however, is that upon reaching the error or staleness bound, the *maximum* key error is known at the center, but there is no further information available for the error of individual keys. Here we briefly describe two alternative approaches that address this disadvantage, albeit with disadvantages of their own.

One alternative is to send a single approximate result for each window, rather than a collection of keys and corresponding approximate results. For example, for a query that performs a `Count` aggregation, we could use a Count-min Sketch [86]. Rather than providing a deterministic bound on the error for each key, this approach guarantees that each key's error is within a factor of ϵ of the true count with probability δ for user-specified parameters (ϵ, δ). While this approach can make more efficient use of constrained WAN bandwidth than our sampling approach, it has its own disadvantages. For one, such approximate data structures are only available for a limited set of queries. For example, if we are computing a `Count` aggregation in order to allow directly querying the count for each key, then the Count-min Sketch approach applies, but if we are computing counts in order to compute the top-K most frequent keys, then we need an alternative data structure, as the Count-min Sketch does not allow us to enumerate keys. In other words, this alternative approach should be considered an optimization that we can apply only in special cases. In practice, such an optimization may best be applied by a higher-level layer responsible for "compiling" application-level queries into runtime execution plans.

Another alternative is to send a probabilistic value for each key. For example, rather than sending a single integer value to represent a count, we could send a more informative statistic such as a confidence interval or an estimate of the probability distribution of the count. Using this approach, each value embeds an estimate of its

own error. But while added error information may be an advantage, this approach may require larger representations for each value, which only exacerbates the challenge of constrained WAN bandwidth. In order to maintain low staleness, it remains necessary apply some sort of sampling approach, such as the prefix-sampling approach we have considered throughout this chapter.

In fact, probabilistic values fit entirely within our model. As long as the user-defined error function appropriately incorporates the value-level approximation, our techniques handle such values directly. Indeed, our experiments demonstrated how our techniques can apply to approximate values, as we used the HyperLogLog sketch [65] to approximate unique counts.

5.6.2 Coordination Between Multiple Edges

We have focused in this chapter on algorithms for trading off staleness and error with one edge and a center. In a multi-edge setting, if keys are partitioned across edges, and if bandwidth out of the edges is the bottleneck, then our techniques apply directly. If, however, each key may arrive at multiple edges, then some coordination between edges is necessary.

Under an error constraint, such coordination is required to "allocate" error across multiple edges. While this is trivial for some aggregation functions (e.g., `Max`), it is challenging in general because error is additive across edges. Ideas from distributed function monitoring [61, 88] may serve as a starting point for developing coordination techniques. Under a staleness constraint, coordination is required to effectively prioritize keys, as it is impossible to compute accumulated error for a key at a single edge if that key has arrived at multiple edges.

Additionally, if the WAN bottleneck arises due to limited bandwidth into the center rather than out of the edges, then it becomes necessary to coordinate the use of this bandwidth across the several edges in order to achieve the best performance. Again techniques from distributed function monitoring may be useful.

5.6.3 Incorporating WAN Latency

We have focused on WAN bandwidth scarcity as the primary cause of staleness, though latency also plays a role. Under a staleness constraint, if latency is non-negligible relative to the staleness constraint, then we should consider this latency. If we do not, then we risk wasting scarce WAN bandwidth sending updates that will only be ignored downstream as they will arrive after the staleness tolerance has been exhausted.

One approach to doing so is to monitor the WAN latency and predict the latency L for the current window. Then, we can apply our algorithms with staleness bound $S - L$ where S is the application-specified staleness constraint.

Note that there is no need to consider WAN latency directly for our error-bound algorithms, as these algorithms reach the error constraint as quickly as possible regardless of the latency in the wide-area network.

5.7 Related Work

Our work bears some resemblance to anytime algorithms (e.g., [89, 90]), which improve their solutions over time and can be interrupted at any point to deliver the current best solution. While our staleness-bound algorithm is essentially an anytime algorithm, terminating when the staleness limit is reached, our error-bound algorithm terminates only when the error constraint is satisfied.

JetStream [12] considers wide-area streaming computation, and like our work, addresses the tension between timeliness and accuracy. Unlike our work, however, it focuses at a higher level on the appropriate abstractions for navigating this tradeoff. Meanwhile BlinkDB [82] provides mechanisms to trade accuracy and response time, though it does not focus on processing streaming data. Our focus is on specific policies to approach an optimal tradeoff between timeliness and accuracy in streaming settings. Das et al. [83] consider tradeoffs between throughput and latency in Spark Streaming, but they focus on exact computation and consider only a uniform batching interval for the entire stream, while we consider approximate computation and address scheduling on a per-key basis.

Our techniques complement other approaches for approximation in stream computing. For example, our work applies to approximate value types such as the Count-min

Sketch [86] or HyperLogLog [65], while recent work in distributed function monitoring [88, 61] could serve as a starting point for coordination between multiple edges.

5.8 Concluding Remarks

In this chapter, we considered the problem of streaming analytics in a geo-distributed environment. Due to WAN bandwidth constraints, applications must often sacrifice either timeliness by allowing *stale* results, or accuracy by allowing some *error* in final results. Focusing on windowed grouped aggregation, an important and widely used primitive in streaming analytics, we studied the tradeoff between the key metrics of staleness and error. We presented optimal offline algorithms for minimizing staleness under an error constraint and for minimizing error under a staleness constraint. Using these offline algorithms as references, we presented practical online algorithms for effectively trading off timeliness and accuracy in the face of bandwidth limitations. Using a workload derived from a web analytics service offered by a large commercial CDN, we demonstrated the effectiveness of our techniques through both trace-driven simulation as well as experiments using an Apache Storm-based prototype implementation deployed on PlanetLab. Our results showed that our proposed algorithms outperform practical baseline algorithms for a range of error and staleness bounds, for a variety of aggregation functions, and under varied network bandwidth and workload conditions.

Chapter 6

Challenges and Opportunities in Unified Stream & Batch Analytics

6.1 Introduction

While both batch processing and stream processing are important topics on their own, applications are increasingly being developed that do not fit neatly into either of these categories. For example, a network operator might rely on up-to-the-second summary data to identify potential performance anomalies, but then wish to compare this data against historical statistics. This application requires synthesizing results computed from both unbounded streams and bounded batches of data. As another example, an analyst who regularly computes a daily summary of an important dataset may discover that this report provides a useful signal for day-to-day decisions, and wish to accelerate the pace of reporting, generating hourly—or even continuous—reports. In other words, an application might evolve over time from processing bounded batches of data to processing an unbounded stream of data in near-real-time. In short, modern applications blur the line between stream and batch analytics.

In order to support these applications, systems need to provide a *programming model* that treats analytics over unbounded streams and bounded data sets in a unified manner.

In addition, they need to provide an *execution engine* capable of satisfying application requirements in terms of timeliness, accuracy, and cost. These programming model and execution engine challenges are rapidly gaining research attention. One key insight from this research is that application semantics are orthogonal to the execution engine. For example, a stream can be discretized into "micro-batches" to be processed by a batch execution engine (e.g., Apache Spark Streaming [60]), and a batch dataset can be viewed as a finite stream and processed by a stream execution engine (e.g., Apache Flink [17] or Apache Apex [18]). It is therefore useful to consider programming models and execution engines separately.

In terms of programming models, Apache Beam [16] (originally Google Dataflow [15]) has emerged as the current standard. The Beam[1] model is a powerful one, allowing concise expression of a broader set of applications than we have studied so far in this thesis. For example, Beam allows more general windowing strategies and is designed from the ground up to address the real-world concerns of out-of-order input data. Although Beam is a powerful model, it has several drawbacks that, while important in a single-datacenter environment, are magnified by the challenges of geographic distribution. Specifically, the abstractions provided by the Beam model are *burdensome* for applications that value tradeoffs along the key dimensions of timeliness, accuracy, and cost. Rather than allowing application developers to explicitly express their constraints for these metrics, Beam forces application developers to describe *how* to carry out computation, in particular by specifying when to materialize results. In addition, Beam's abstractions are *insufficiently expressive* to allow application developers to achieve their desired tradeoffs between timeliness, accuracy, and cost.

In terms of execution engines, reliable stream processing systems are a popular focus of research and engineering effort in recent years. The techniques presented in this thesis are useful for more systematically optimizing timeliness, accuracy, and cost in these systems, but there are many opportunities to extend these techniques in order to more fully address the broader class of applications enabled by the Beam model.

In this chapter, we highlight several opportunities and challenges at the frontier of both programming models and execution engines. Through trace-driven simulation using our Akamai workload, we demonstrate the limitations of the Beam model in terms

[1] Batch + stream

of both convenience and expressiveness. We show that the techniques from Chapters 4 and 5 can help navigate the timeliness-accuracy-cost tradeoff space more systematically than common practical approaches, even in the face of Beam's current limitations. We show that, if Beam were extended to overcome some of its limitations, our techniques could be even more effective. Finally, we highlight directions for extending our techniques that could enable more effective timeliness-accuracy-cost tradeoffs for the broader class of applications that Beam supports.

6.2 Background

6.2.1 Diverged Stream and Batch Processing

Historically, researchers and engineers have considered batch processing and stream processing to be separate problems, and distinct systems have been developed and optimized for each. At one extreme lie batch processing systems such as Google MapReduce [13] and its open-source Hadoop [24] counterpart. These systems were originally designed to process massive data sets on very large clusters of commodity hardware. In these settings, jobs may run for many hours, and hardware failures are the common case rather than the exception. In order to ensure progress in the face of failures, these systems rely heavily on persisting state to disk, both locally and in replicated distributed file systems such as GFS [91] and HDFS [92]. As a result, these systems promise reliable computation, but do so with latency on the order of minutes or hours.

At the other extreme lie stream processing systems such as Apache Storm [58]. Unlike batch processing systems which process a bounded set of input data, these systems execute long-running jobs that process unbounded streams of data in a record-by-record manner. By default, Storm provides only at-most-once processing semantics: some records may fail to be fully processed due to task failures. To address this weakness, Storm provides an acknowledgement mechanism that allows programmers to replay records when they may not have been fully processed. While this is useful for achieving at-least-once semantics in many cases, it is not always easy to apply, for example when applications naturally batch records to compute windowed aggregation. Achieving exactly-once semantics in Storm is complex enough that a separate API is provided for this use case [93]. Overall, early stream computing systems such as Storm promise

low-latency results, but typically do so with looser accuracy guarantees than batch systems.

6.2.2 Early Attempts at Unified Stream & Batch Processing

Early attempts at unifying stream and batch processing have fallen short for various reasons. One of these early attempts is the so-called Lambda architecture [94], where the idea is to simultaneously run two versions of every application, one using a reliable but high-latency batch processing system, and the other using a low-latency but possibly approximate stream processing system. The results from these systems are merged at query time, with the streaming results serving as a stop-gap until the batch processing system produces more reliable results. The major downside of this approach is that it requires deploying and managing two processing systems, and developing and maintaining two versions of every application. There have been some attempts to mitigate these downsides, for example by compiling the same high-level program into both batch and stream versions [59], but the operational inefficiencies remain.

The Kappa Architecture [95] was proposed in response to these weaknesses, and it rejects the notion that stream processing systems are inherently approximate or less powerful than batch processing systems. In the Kappa Architecture, all computation is carried out using a stream processing system, and simultaneous versions of an application run only when version $n + 1$ of an application is replaying old inputs and catching up to the results produced by its predecessor, version n. At its core, the Kappa Architecture is more a rejection of the Lambda Architecture than a concrete proposal for a new approach.

Yet another approach, embodied by Spark Streaming [60], is to model an unbounded stream as a sequence of bounded batches, and apply an efficient batch processing system to each of these batches. One benefit of this so-called micro-batching approach is that the same infrastructure—and largely the same programming model—can be employed for both stream and batch computing. One major weakness, however, is that this approach tightly couples application-level windowing with execution-engine-level batching. This is especially problematic for applications where records must be grouped into time windows based on an externally-generated timestamp (i.e., *event time*) rather than the time at which the system processes them (i.e., *processing time*).

6.2.3 The State of the Art: Apache Beam

The state of the art is trending toward a unified programming model capable of expressing both batch and stream programs [15], as well as unified execution engines capable of efficiently computing over both bounded batches and unbounded streams [17, 18]. One leading model is that of Apache Beam [16]. This model is motivated by two key ideas: first, that batch processing can be viewed as a special case of stream processing; and second, that stream processing applications must explicitly confront the fact that inputs may arrive out of order, and as a result, the completeness of inputs can never be fully guaranteed.

The Beam model requires programmers to answer the following questions.

What results are calculated; i.e., which *transformations* are applied?

Where in event time do window boundaries lie?

How are results refined as additional (possibly late) data arrive?

When in processing time are results materialized?

In terms of the first question (*what* transformations to perform), Beam allows more than just the aggregation we have considered in the previous two chapters. For example, Beam applications may include mapping, filtering, or joining operations. These other transformations present interesting challenges, such as determining where to place computation (as discussed in Chapter 2) or how to best translate applications into runtime execution plans (e.g., WANalytics [8]). Here we focus on the rich challenges that have not yet been explored in the context of aggregation.

Beam also allows more general windowing, as captured by the second question (*where* window boundaries lie). In addition to the tumbling windows we have explored so far, Beam also allows concise expression of hopping windows—discrete approximations of continuously sliding windows—as well as session windows among others. Session windows raise entirely new challenges and are worthy of significant research effort, though we focus here on the more immediate extension of our techniques to hopping windows.

In many real-world applications, it is crucial to window inputs by their event-time—i.e., the time at which inputs are generated by an external system or user—rather than

the time at which they arrive at or are processed by the execution engine. In general, whether due to intermittent connectivity between data sources and edges, heterogeneous network performance, failure of intermediate systems, or any number of other factors, inputs may arrive out of order with respect to this event time. Some arrivals are sufficiently delayed relative to others that they may be deemed "late". The choice of how to deal with such late arrivals is the focus of Beam's third question: *how* refinements relate. This policy is largely dictated by the application: Only the application developer can answer whether updated results should replace previous results or instead augment them. Here we explore some of the lower-level challenges of processing data that do not arrive in strict event-time order.

While each of the above questions presents interesting challenges, it is the final question—*when* to materialize results—that raises the richest research challenges in geo-distributed settings. This question maps closely to questions we have already explored in this thesis, specifically when to communicate partial results (Chapters 4 and 5) and precisely what results to communicate (Chapter 5). As we have shown, the answer to the question of when to materialize results—the *triggering* policy in Beam parlance—has a significant impact on timeless and accuracy of results. For example, if an application requires results with low staleness, but can tolerate greater error, then results should generally be materialized earlier. On the other hand, if an application demands a high degree of accuracy but can tolerate staleness, then it is advantageous to produce results later, when a more complete view of inputs is available. In that sense, the choice of when to materialize results is an important tuning knob that Beam application developers can use to influence the timeliness-accuracy tradeoff.

Beam Triggers

Throughout this chapter, we will study the impact of different triggering policies, so it is useful to understand more precisely the Beam trigger abstraction. It is the role of a triggering policy to determine, for a given key in a given window, when to materialize results. The policy is invoked upon the arrival of new data for that key and periodically as both processing time and event time advance. The result of the invocation is a binary

decision whether to materialize results or continue accumulating.[2]

Note that triggers are unable to consider the state of other keys or windows, or to consider the broader runtime state such as recent data arrival patterns or outgoing network performance. As we will show throughout this chapter, these are significant limitations for applications that value performance along the key dimensions of timeliness, accuracy, and cost.

6.2.4 Challenges Magnified by Geo-Distribution

Continuing the pattern from the previous chapters, we consider a hub-and-spoke architecture (see Figure 4.1), comprising one or more edges connected to a single central location by a bandwidth-constrained wide-area network (WAN). To run in such a geo-distributed setting, a Beam application needs to be decomposed into edge and central components, each of which can be expressed as a Beam application. To handle the full generality enabled by Beam, it would be the role of a query planner to determine *what* transformations to apply at each location. We focus on applications that perform windowed grouped aggregation and are decomposed such that partial aggregation can be performed at the edge(s), and final aggregation performed at the center. Likewise, the policy for *how* refinements relate is dictated by the application, and we consider this to be another task for an application-level query planner. The question of *where* window boundaries lie in event time is unaltered in the process of decomposing applications across the wide-area.

The triggering question, that of *when* to materialize results, is perhaps the most difficult and the most critical in geo-distributed settings, as it has a profound effect on timeliness, accuracy, and cost, as we have shown throughout this thesis. Effectively optimizing performance along these dimensions requires a sophisticated answer to this question, and we argue that application developers should not be responsible for deriving such a policy. Instead, rather than specifying the triggering policy, application developers should specify their requirements along the dimensions of timeliness, accuracy, and

[2] More precisely, the policy determines both whether to fire and whether to retain any accumulated results upon firing, but the latter question is determined by the application policy for how result refinements relate.

cost, and the system should accept responsibility for employing an appropriate triggering policy to meet these requirements. Throughout this chapter, we explore challenges both on the programming model side, and in terms of the techniques that execution engines can use to make effective triggering decisions.

6.3 Simulation Setup

Throughout this chapter, we reuse the simulation framework first introduced in Section 4.5. This simulator models an aggregation algorithm—including its triggering policy—as a function from a sequence of incoming arrivals to a sequence of outgoing updates. Traffic is determined by the number of such updates, while error is computed by aggregating these updates and comparing against an exact aggregation of the original arrival sequence. Staleness is computed by simulating the network as a queueing system with deterministic service times based on the network bandwidth.[3]

Throughout this chapter, we use the anonymized Akamai trace with the `large` query key (see Section 4.3). Except where we show time series, figures reflect results over sixty windows representing multiple days' worth of arrivals drawn from the busiest portion of the trace. Except where otherwise noted, we simulate a medium-bandwidth WAN, which corresponds to sufficient bandwidth to maintain bounded staleness for exact aggregation over tumbling windows when using a traffic-optimal algorithm, but not when using a pure streaming algorithm.

As in earlier chapters, for confidentially reasons, we show staleness values normalized relative to the window length, traffic values normalized relative to pure streaming, and error values normalized relative to the maximum possible error for the given aggregate over the portion of the trace used for our simulations.

6.4 Programming Models

While the question of when to materialize—and in turn, flush—results is important, there are two problems with requiring application programmers to answer this question.

[3] We also simulate the network using exponential service times, and no conclusions discussed in this chapter are altered by this change. We report results using the deterministic model for consistency with the previous chapters.

First, it is burdensome. Application developers are concerned with their high-level accuracy, timeliness, and cost requirements, and not with low-level scheduling concerns. Further, this question cannot be answered effectively in a simple straightforward manner. As we have seen in Chapters 4 and 5, an effective answer to this question must consider dynamic network and data characteristics, as well as application requirements. It is therefore unreasonable to require application developers to answer this question.

Second, the current Beam triggering abstractions are not expressive enough to effectively navigate the timeliness-accuracy-cost tradeoff space. By forcing triggers to consider only a single key and window at a time, and to make their decisions without awareness of runtime conditions such as WAN network congestion, Beam precludes the expression of powerful triggering policies such as our two-part caching algorithms.

6.4.1 Convenience

Application developers ultimately care about timeliness, accuracy, and cost, and they should be allowed to directly specify their requirements along these dimensions. Without this, application developers are burdened with a process of trial and error to find an appropriate triggering policy to meet their requirements. While this is a powerful tuning knob [15], it should not be the responsibility of application programmers to carry out this exploration. Instead, systems should accept the responsibility for automatically determining an appropriate triggering policy given user-specified requirements for timeliness, accuracy, and cost.

To demonstrate the burden imposed by the current Beam model, consider an application that performs a `Sum` aggregation over tumbling windows with the goal of minimizing staleness subject to an error constraint. We consider a range of triggering policies for this application.

Streaming: This policy immediately materializes results for a key upon the arrival of a new record; i.e., it streams arrivals directly to the center. This reflects an approach that fails to utilize the edges for aggregation.

Batching: This policy only materializes (and flushes) aggregate results for a key when the end of the window is reached. This pure batching approach reflects the default triggering policy in Beam.

Micro-batching: This policy materializes results more frequently, specifically eight times per window. This effectively emulates a micro-batching execution engine such as Spark Streaming [60], where unbounded streams are discretized into small batches, each of which is processed by a batch processing engine. Computing final results requires aggregating across micro-batches (eight in our case).

This also reflects a common approach a Beam application developer might employ in order to reduce latency. The idea is to reduce the delay in transmitting the final results for the window by flushing smaller batches more frequently, as opposed to flushing a single large batch at the end of the window.

Lazy: This policy, which we will use throughout the chapter, is inspired by the Eager Prefix Error (EPE) offline optimal algorithm for error-constrained computation introduced in Section 5.3.1. The Lazy algorithm[4] is designed to be implementable in Beam, so it makes triggering decisions based on each individual key and window in isolation. Specifically, it maintains the necessary state to compute the accumulated error (see Section 5.4.2) for the key, and it notes the elapsed time between the beginning of the window and the first time at which the accumulated error exceeds the error constraint E. Let t_i denote this elapsed time. At this point, we know that omitting the key would exceed the error constraint, so it must eventually be flushed. The Lazy algorithm makes the assumption that the future mimics the past, and that error will continue to accumulate at the same rate throughout the remainder of the window. Based on this assumption, the key is scheduled to be flushed at time $t_f = T + W - t_i$, where $T + W$ denotes the end of the window.[5] Until time t_f, arrivals are aggregated for the key, and at time t_f the key is flushed. After time t_f, the key is flushed again if and when accumulated error exceeds the error constraint, which can occur zero or more times depending on the actual arrivals for the key.

The reason we call this the Lazy algorithm is simply because it aims to delay the first flush until it predicts that the key has accumulated all but E of its final value, and is therefore ready to be flushed.

[4] Note that, although we are reusing the name, this is not the same algorithm as the Lazy algorithm for exact aggregation in Section 4.4.

[5] If $t_i \geq \frac{W}{2}$, then the key is flushed immediately at $t_f = T + t_i$.

Caching: This policy reflects our two-part caching online algorithm discussed in Section 5.4.2. It uses a value-based cache partitioning policy and an LRU cache eviction policy to emulate the EPE offline optimal algorithm. We must emphasize that while this is an online algorithm, it cannot be implemented using the current Beam abstractions, because it determines when to materialize results based on a view of all keys, rather than a single key in isolation. Additionally, it utilizes an awareness of the outgoing WAN bandwidth to determine how rapidly to flush keys.

We include this algorithm to study the additional benefit we might realize if Beam were to offer more powerful triggering abstractions.

Offline: Finally, the offline optimal algorithm denotes the EPE algorithm itself, which requires complete knowledge of future arrivals for each key. This algorithm cannot be implemented in an online setting, but is included here to help contextualize the other practical algorithms.

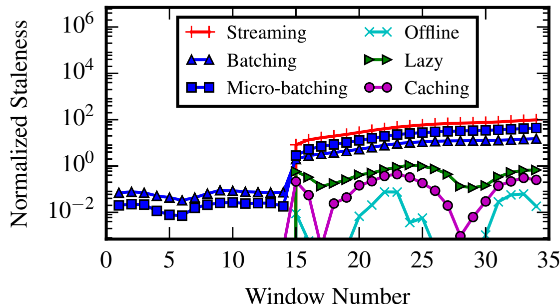


Figure 6.1: Staleness over time for a `Sum` aggregation under an error constraint.

Figure 6.1 shows a time series of staleness for this set of algorithms for our error-constrained `Sum` aggregation. Here we use a low WAN bandwidth to more clearly highlight our points. We see that neither Streaming nor the Batching variants (Batching and Micro-batching) consistently perform well. Streaming achieves very low staleness (close to zero, and hence not visible given the logarithmic y-axis scaling of the figure) for the first portion of the window, but excessive staleness when the workload shifts and it is no longer feasible to stream updates to the center. The Batching variants yield lower—though still excessive—staleness during the latter portion of the simulation, but do so

at the cost of higher staleness in the earlier windows. Even between the two Batching algorithms, there is no clear better choice. Micro-batching yields lower staleness during the early windows, but higher staleness in the later windows due to its higher traffic. In short, these simple algorithms, although easily expressed in Beam, are insufficient to achieve desirable performance in terms of timeliness, accuracy, and cost.

To better navigate this tradeoff space, a Beam application developer needs to implement her own trigger policy, and our Lazy algorithm reflects a principled approach to this task. By exploiting our application's tolerance for error, this algorithm yields very low staleness in the early windows, and staleness that remains below one window length in the latter windows. This is a dramatic improvement over the naïve Batching, Micro-batching, and Streaming algorithms, but implementing such a policy is nontrivial, and application developers should not be burdened with this responsibility.

Our Caching algorithm yields even better performance than the Lazy algorithm, demonstrating that a richer triggering abstraction could yield more effective timeliness-accuracy-cost tradeoffs than possible under the current Beam model.

6.4.2 Expressiveness

For error-constrained applications, we saw that Beam's per-key, per-window triggers limit our ability to express triggering policies that effectively trade timeliness, accuracy, and cost. For applications with a constraint on staleness and a desire to minimize error, the limitations of the Beam model are more severe: the current abstractions *completely* limit the ability to specify practical triggering policies for minimizing error under a staleness constraint. Specifically, in a staleness-constrained case, the choice of when to materialize and flush results for any given key can only be made in the context of the other keys, and with an awareness of the outgoing wide-area network. The lack of cross-key triggers is therefore a severe limitation of the Beam model.

Given Beam's current limitations, one possible approach for staleness-constrained applications is to use an error-bound algorithm and attempt to set the error bound as low as possible without violating the staleness bound. This is, however, a poor solution, as it requires trial and error to determine an appropriate error bound, and in fact it is not possible to set a static error bound that guarantees satisfying the staleness bound.

To illustrate this point, we simulate an optimal offline error-bound algorithm with

three error constraints (0, Lo, and Hi). We use an offline optimal algorithm rather than an online algorithm to focus on the limitations of using an error-bound algorithm in general rather than on the particular behavior of a specific online error-bound algorithm. We also simulate an offline optimal *staleness-bound* algorithm (see Section 5.3.1) to demonstrate the performance potential of a true staleness-constrained algorithm. We again use a low-bandwidth WAN to highlight our points.

Figures 6.2 and 6.3 show staleness and error, respectively, for these four algorithms, for a `Sum` aggregation with a staleness tolerance of $\frac{W}{2}$.

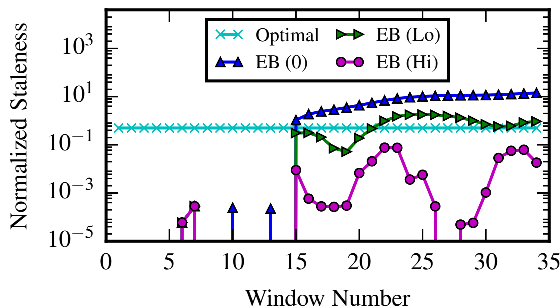


Figure 6.2: Staleness over time for a `Sum` aggregation under a staleness constraint.

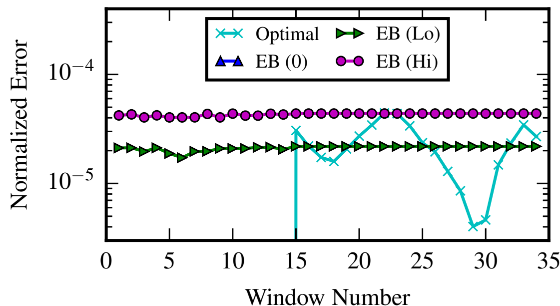


Figure 6.3: Error over time for a `Sum` aggregation under a staleness constraint.

From these figures, we observe that even exact computation (i.e., the EB (0) algorithm) suffices for the first several windows. In the latter portion of the workload, however, it is necessary to tolerate more error in order to maintain the staleness constraint, but the necessary error tolerance varies from window to window. For some

windows, the low error tolerance is enough to satisfy the staleness constraint, while other windows exceed the staleness constraint even with the high error tolerance.

A genuine staleness-bound algorithm, on the other hand, consistently guarantees the staleness constraint for each window, and minimizes the error given that constraint. In short, we can see that emulating a staleness-bound algorithm by applying an error-bound algorithm yields poor results and fails to achieve the desired performance along the dimensions of timeliness, accuracy, and cost.

While Beam is a useful and powerful model, the triggering abstraction is insufficiently expressive for applications in which the timeliness-accuracy-cost tradeoff is important. To reliably satisfy a staleness constraint while minimizing error, triggers need to be able to prioritize across keys, and to be informed not just by details of the arriving data, but also by outgoing network conditions.

6.4.3 Research Challenges

As we have shown, the current Beam abstractions are both burdensome and insufficiently expressive for applications that value performance along the key dimensions of timeliness, accuracy, and cost. Instead of asking application developers *when* to materialize results, Beam should allow users to specify their requirements for these key metrics. It should be the responsibility of systems to determine when to materialize results. To make effective decisions, these systems need the ability to make triggering decisions not just for isolated keys, but across keys, and with additional visibility into runtime information such as outgoing network conditions.

To effectively make triggering decisions, systems ultimately need effective algorithms, which we have presented in the previous chapters and study further in the remainder of this chapter. To support these algorithms, systems need to provide more powerful mechanisms for triggering. Allowing cross-key and network-aware (and as we will show later, cross-window) triggering would enable algorithms that better navigate the timeliness-accuracy-cost tradeoff space. These more powerful mechanisms do, however, raise systems challenges in terms of scalability. The current triggering model allows each key to be logically independent of all other keys, which simplifies tasks such as maintaining and partitioning state, as well as defining and scheduling tasks across clusters of machines. Providing triggers with a broader set of inputs makes these matters

more challenging; novel techniques may be required to communicate this information efficiently, or to reduce the scope of required coordination.

6.5 Execution Engines

While there are challenges that need to be addressed in the Beam model, the added expressiveness highlights directions in which our algorithms may also need extension. In particular, we focus here on the implications of more general windowing and out-of-order arrivals. This exploration highlights the extent to which the techniques in this thesis advance the state of the art, while at the same time highlighting challenges that require further research.

6.5.1 Hopping Windows

Throughout this thesis, we have focused on tumbling windows; i.e., non-overlapping windows of length W. While tumbling windows are common and useful, there are other interesting window types to consider.

A more general type of window is the *hopping window*, as shown in Figure 6.4. Such a window has length W but advances at interval H, where W is a multiple of H. As an example, we might use hopping windows to compute an aggregate over 1-hour windows, beginning a new window every 10 minutes (i.e., $W = 1\,\text{hour}$ and $H = 10\,\text{minutes}$). Intuitively, as the ratio W/H increases, hopping windows more closely approximate sliding windows. In the limit as W/H approaches infinity, windows are continuously sliding.

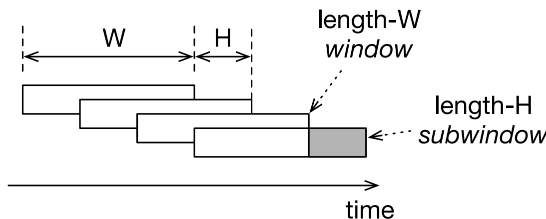


Figure 6.4: Hopping windows have length W and advance at interval H.

We refer to such windows as (W, H) hopping windows throughout this section. We use *window* to refer to the full length-W window, while *subwindow* refers to one of the

W/H contiguous length-H subparts of each window. Note that, in general, hopping windows overlap.

When computing hopping windows, updates flushed by the edge(s) must somehow encode the windows to which they belong. In the tumbling window case, it suffices to flush (time, key, value) triples, where time corresponds to any time within the specific window. Now, because windows overlap, using a single time would allow an update to apply to multiple windows. To unambiguously specify the windows to which an update should apply, we make this timestamp two-dimensional, indicating the *time interval* over which the update applies. Concretely, updates take the form $([t_s, t_e), k, v)$ indicating that they apply to all windows $[T, T + W)$ where $T \leq t_s$ and $t_e \leq T + W$. The center then applies each incoming update to all windows containing the interval $[t_s, t_e)$.

Exact Computation

To demonstrate the added challenge of hopping windows, we begin by exploring exact computation. We consider three algorithms.

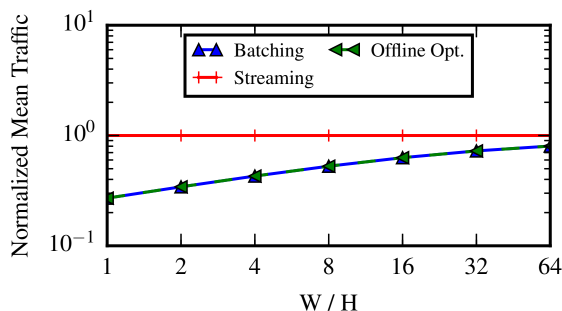
Streaming: A pure streaming approach logically performs no work at the edge, and instead merely forwards arrivals directly to the center.

Batching: For exact computation, we must flush updates reflecting *all* arrivals. Therefore, for every key that has arrivals in a particular length-H subwindow, an update must be flushed. In other words, it is not possible to send any less traffic than an algorithm that simply applies pure batching within each length-H subwindow. This is exactly the approach taken by our Batching algorithm.

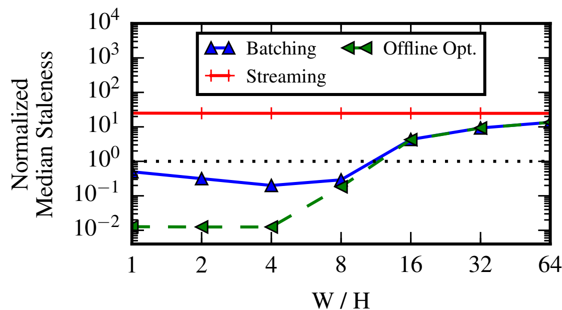
Optimal: While the Batching approach is optimal for traffic, its main disadvantage is that it defers all communication until the end of each length-H subwindow, which can lead to high staleness. To mitigate this problem, a more effective algorithm flushes updates prior to the end of the window. In fact, when exact results are required, the problem of computing (W, H) hopping windows is equivalent to that of computing length-H *tumbling* windows at the edge and relying on the center to logically concatenate these subwindows to form full length-W windows. We

can therefore apply the techniques from Chapter 4 to each of these length-H sub-windows. Specifically, our Offline Optimal algorithm applies the offline optimal algorithm for exact computation from Section 4.4.

While matching the optimal traffic of a pure batching approach, this offline algorithm also achieves optimal staleness by flushing updates for keys that have reached their final value without waiting until the end of the window.



(a) Traffic normalized relative to Streaming.



(b) Staleness normalized by the window length.

Figure 6.5: Traffic and staleness for an exact `Sum` aggregation.

Figure 6.5 shows traffic and staleness for these algorithms as W remains fixed and H decreases, yielding a higher W/H ratio. Note that in this figure and throughout the remainder of the chapter, we simulate a medium-bandwidth WAN. These results illustrate several key points. For one, Streaming exhibits the highest traffic, and given the network capacity in this simulation, staleness that quickly exceeds one window length

and continues to grow throughout the duration of the simulation. This demonstrates the importance of utilizing both edge and central resources to minimize traffic and staleness.

Further, as W/H increases, optimal traffic increases. The reason is that, as H decreases and we approach sliding windows, aggregation is performed over shorter (sub)windows, and is therefore less effective in reducing the number of flushed updates relative to the number of arrivals.

Optimal staleness also increases with W/H due to the increasing traffic. When W/H is sufficiently high, traffic exceeds the network capacity, and as a result, delays accumulate from window to window, leading to very high staleness. For a pure batching algorithm, staleness first decreases before increasing and eventually converging to that of the optimal algorithm. The reason for this non-monotonic trend is that, while total traffic increases with W/H, the size of each subwindow's batch decreases, and the delay in transmitting the updates for each window therefore decreases as well. Once traffic increases sufficiently, however, congestion becomes a more persistent concern and staleness quickly grows as a result. At high W/H, traffic exceeds the network capacity for this simulated network, and both the offline optimal algorithm and a pure batching algorithm owe their high staleness to network congestion rather than the fine-grained details of when each key is flushed.

Overall, we see that as W/H increases, the effects of a constrained WAN become more pronounced, and jointly minimizing traffic and staleness becomes increasingly challenging. For applications requiring exact, near-real-time results in the face of constrained WAN capacity, it is not feasible to approximate sliding windows by arbitrarily increasing W/H. Instead, applications must make some compromise, either by settling for coarse hopping windows or pure tumbling windows, or by allowing approximation.

Error-Constrained Computation

In many cases, exact near-real-time computation is infeasible, and applications are willing to tolerate some degree of error. When this approximation is permitted, hopping windows become significantly more challenging. In fact, even offline optimal algorithms for minimizing staleness under an error constraint are not yet known. In this section we explore the strengths and weaknesses of practical approaches, and motivate further research.

From a high level, we can divide these practical approaches into two classes: *serial subwindows*, and *parallel windows*.

Serial Subwindows. Using the *serial subwindows* approach—*serial* for short—the edges compute length-H tumbling windows and the center logically concatenates results from a series of these windows to form full length-W windows. While this is optimal under exact computation as argued earlier, it is not in general optimal for approximate computation. The primary disadvantage of this approach is that it is necessary to logically divide the error tolerance across each of the W/H subwindows constituting each length-W window.

As a concrete example, consider a `Sum` aggregation with $(W, H) = (10, 5)$ and an error constraint of $E = 100$. In order to guarantee that the error within every length-W window satisfies the error constraint, we need to ensure that the `Sum` of errors for any key across any two adjacent length-H subwindows does not exceed E. The most straightforward way to do this is to enforce an error constraint of $E' = E \cdot \frac{H}{W} = 50$ for each length-H subwindow.[6] The problem, however, is that keys do not necessarily exhibit uniform error across subwindows. For example, it may be possible to achieve lower staleness by allowing one key an error of 70 in one subwindow and only 30 in the adjacent subwindows. Meanwhile, another key may have arrivals only in a single subwindow, in which case the full error tolerance could be allotted to that single subwindow.

In short, the problem with the *serial* approach is that it limits our ability to fully exploit an application's tolerance for error, which results in higher-than-optimal traffic and staleness.

Parallel Windows. Another approach is to compute (W, H) hopping windows by logically running W/H separate length-W tumbling window aggregation algorithms at the edge, with each phase-shifted by $k \cdot H$ for $k = 0, 1, \ldots, W/H - 1$. Because the edge is essentially performing W/H aggregations in parallel, we refer to this as the *parallel-windows* approach, or more concisely the *parallel* approach. Each of these parallel instances is oblivious to the decisions made by the others, so the primary disadvantage of this approach is that traffic grows as W/H increases.

[6] Some aggregations, such as `Max` do not suffer from this problem. We explore the implications of this later.

As a concrete example, consider a key with a single arrival. The parallel approach would flush an aggregate value for this key once for each of the W/H windows that contain the arrival. In a pathological case where every key is unique, traffic would be W/H times that of streaming!

In an offline setting, it is possible to filter these duplicates. Specifically, if each parallel aggregation algorithm runs the EPE algorithm from Section 5.3.1, then duplicates manifest as identical (key, value) updates flushed at the same instant in time. To eliminate duplicates, we can collect all updates flushed by the W/H parallel aggregation algorithms, and group records by departure time, key, and value. Each update indicates the window to which it was originally destined, specifically by denoting the beginning and end time of that window. For each group, we flush a single update that applies only to the interval $[t_s, t_e)$ where t_s is the *largest* start time of any destination window, and t_e is the *smallest* end time of any destination window. In other words, we indicate that each update applies to the *intersection* of intervals contained in the group of duplicate updates. Because the center applies an update for time interval $[t_s, t_e)$ to *all* windows containing $[t_s, t_e)$, this results in applying this single update to exactly the set of windows to which the original group of duplicate updates would have applied.

It is worth emphasizing that this technique only applies in offline settings, as it relies on identifying duplicates based not just on identical keys and values but also identical departure times, as these three dimensions guarantee perfectly redundant updates given the definition of the EPE algorithm.

Further, although this approach avoids the blatant duplication of the baseline parallel approach, it is not itself optimal. To illustrate why, consider a simple example, focusing on a single key, and assume a simple `Sum` aggregation. Figure 6.6 shows four arrivals, at times t_i with values v_i for $i = 1..4$.

The true aggregate values for windows 1 and 2 are $v_1 + v_2 + v_3$ and $v_2 + v_3 + v_4$, respectively. Assume the error constraint is E, and that all four individual values are smaller than E, but that the sum of any two values is larger than E. That is, an error-constrained algorithm must flush updates that reflect at least two of the three values arriving within each window.

The offline optimal EPE algorithm essentially identifies the shortest *prefix* of arrivals that cumulatively satisfy the error constraint for each window. As a result, it would

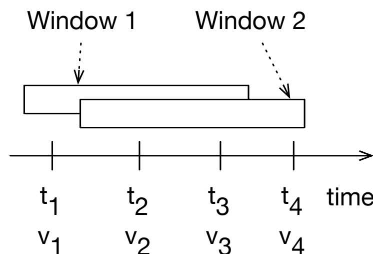


Figure 6.6: The parallel approach with (offline) duplicate elimination is suboptimal.

flush a value of $v_1 + v_2$ at time t_2 for window 1, and a value of $v_2 + v_3$ at time t_3 for window 2.

An alternative approach is to aggregate a *suffix* of arrivals for window 1 and a prefix for window 2, and emit a single update with value $v_2 + v_3$ at time t_3. Because both v_1 and v_4 are smaller than E, this satisfies the error constraint for both windows, yet it reduces traffic by avoiding one redundant update. The effect on staleness depends on other keys, the network capacity, and the actual values of t_2 and t_3, but under some circumstances (for example if $t_3 - t_2$ is small) this could also reduce staleness.

This highlights the added challenge facing algorithms for approximate aggregation under hopping windows: it is no longer sufficient to consider only which *prefix* of values to aggregate for a key. Instead, algorithms must more generally consider which *subsequence* of arrivals to include in their aggregates, and they must do so with consideration given to multiple overlapping windows. Developing such algorithms, however, remains an open research problem.

`Sum` Aggregation. In the meanwhile, although both serial and parallel algorithms are known to be suboptimal, they do provide some insight into the additional challenges presented by hopping windows. Toward this end, Figure 6.7 shows traffic, staleness, and error for several serial and parallel algorithms for a `Sum` aggregation under an error constraint.

Parallel/Serial Offline: These algorithms apply our EPE algorithm using the parallel and serial approaches, respectively.

Parallel/Serial Lazy: These algorithms apply the online Lazy algorithm introduced in Section 6.4 using the parallel and serial approaches, respectively. Recall that this algorithm is inspired by the EPE algorithm, and it makes its triggering decisions with knowledge of only time and the arrivals for a single key. In other words, it is possible to implement this algorithm using the current Beam abstractions.

Serial Caching: This algorithm applies our two-part caching techniques from Section 5.4.2 using the serial approach. Note that, because this algorithm was designed explicitly for disjoint windows, we simulate only the serial approach. Although this is an online algorithm, it is not possible to implement it using the current Beam abstractions, as it makes decisions—in particular cache eviction decisions—across keys and with network awareness. We include it here to further motivate the addition of more robust triggering mechanisms to modern execution engines.

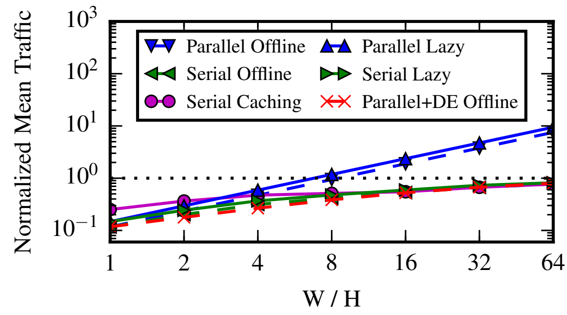
Parallel+DE Offline: This algorithm applies the EPE algorithm with the offline duplicate elimination technique. Although still not optimal, this demonstrates the potential of an algorithm that makes decisions across keys to reduce traffic.

From a high-level, these results reinforce our conclusions for exact computation: as W/H increases, staleness and traffic also increase. We also see that, for all but the smallest values of W/H, the parallel approach performs poorly due to its excessive traffic and in turn high staleness.

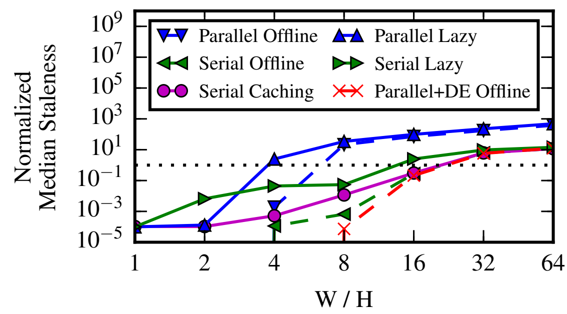
The Lazy algorithms achieve traffic close to their optimal counterparts, while our two-part caching approach shows slightly higher traffic at low W/H, as that algorithm deliberately aims to fully utilize the available network capacity during each window. As W/H reaches roughly 16,[7] this gap vanishes, as even the offline optimal algorithm produces enough traffic to fully utilize our simulated network.

In terms of staleness, Figure 6.7(b) shows that even the relatively simple Lazy algorithm can be effective when using the serial approach. Our two-part caching approach, however, offers significantly lower staleness, especially at intermediate W/H. The reason is that it is easier to predict *which key* among many is the best candidate for an

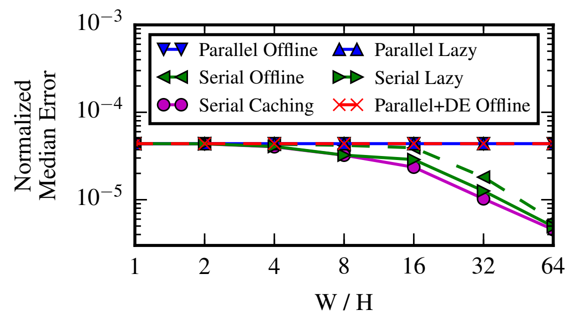
[7] In general, the particular value of W/H at which this occurs depends on the arrival rate, query, the specific value of H, and the WAN capacity.



(a) Mean traffic normalized relative to Streaming.



(b) Median staleness normalized by the window length.



(c) Median error normalized by the maximum possible error for our inputs.

Figure 6.7: Traffic, staleness, and error for a `Sum` aggregation under an error constraint.

early flush than to predict precisely *when each key* should be flushed. This demonstrates again that cross-key triggers would be a powerful addition to Beam.

Figure 6.7(c) quantifies the drawbacks of the serial approach. As W/H increases, the parallel approach is able to fully utilize the error tolerance, while the serial approach utilizes only a fraction of this tolerance, as it uniformly distributes the error tolerance across each window's constituent subwindows.

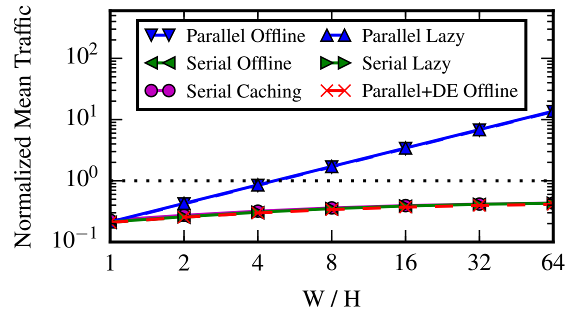
The main problem of both the serial and parallel approaches is that they fail to optimize *across multiple windows*. The serial approach could be improved by considering error across multiple serial windows, while the parallel approach could be improved by removing redundant communication across multiple parallel windows. The offline parallel algorithm with duplicate elimination avoids the drawbacks of both the serial and baseline parallel approaches, yielding the best traffic, staleness and error of the algorithms considered here. As discussed earlier, however, optimal algorithms are not yet known; there remains an opportunity to better optimize across keys and windows to further improve timeliness, accuracy, and cost.

`Max` Aggregation. We next consider an error-constrained `Max` aggregation. One interesting aspect of the `Max` aggregation as opposed to the `Sum` aggregation we have explored so far is that we can allow the full error limit in each of the subwindows. As a result, the serial approach does not suffer from its main drawback. Figure 6.8 shows traffic, staleness, and error using the same algorithms as in the previous figure.

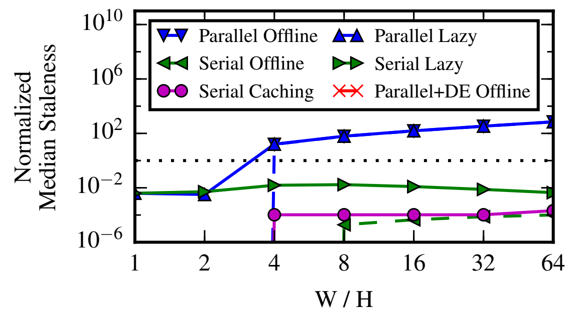
We again observe the same high-level trends, specifically that traffic and staleness increase with W/H. Now, however, both serial and parallel fully exploit the error limit. As a result, the serial approach yields very low staleness. Again, however, the parallel approach with duplicate elimination dominates the other algorithms, as it optimizes across multiple windows. (Staleness is 0 across the range of W/H for the Parallel+DE algorithm, so it does not appear in Figure 6.8(b), which uses a logarithmic y-axis scale.)

Staleness-Constrained Computation

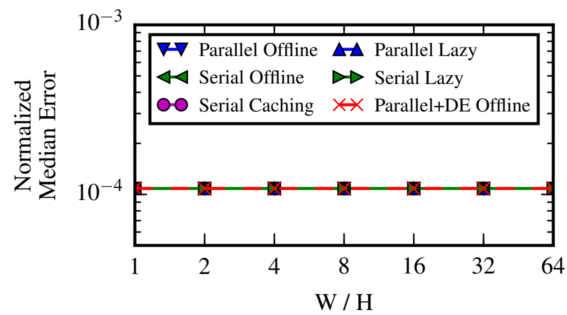
Without cross-key and network-aware triggers, Beam presents severe obstacles to implementing a staleness-bound algorithm. Here we briefly explore how staleness-bound algorithms would perform for hopping windows, for both an offline algorithm and for



(a) Mean traffic normalized relative to Streaming.



(b) Median staleness normalized by the window length.



(c) Median error normalized by the maximum possible error for our inputs.

Figure 6.8: Traffic, staleness, and error for a `Max` aggregation under an error constraint.

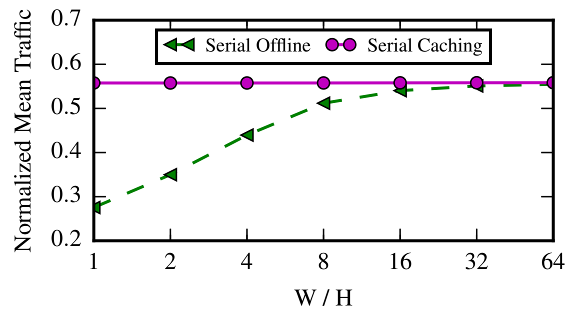
our two-part caching online algorithm from Section 5.4.3. Note that this algorithm, while online, is not implementable in Beam, because it makes both cache partitioning and cache eviction decisions based on information for multiple keys, and with awareness of outgoing WAN performance. Given that both our offline optimal and online two-part caching algorithms assume disjoint windows, we explore only the serial approach. Figure 6.9 shows traffic, staleness, and error for these two algorithms with a staleness constraint of $\frac{W}{16}$.

We see that our serial caching approach maintains consistent traffic across a range of W/H, since this algorithm intentionally fully utilizes the network. The serial offline algorithm, on the other hand, only fully utilizes the network as W/H becomes large; prior to this it is not necessary to fully utilize the network, as aggregation and omission serve to reduce traffic. This difference arises because our staleness-bound algorithm was not designed to minimize traffic. These results highlight an opportunity to improve our staleness-bound algorithms to additionally emphasize traffic reduction.

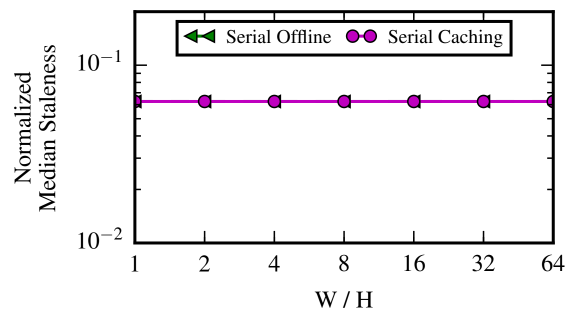
While the caching algorithm shows higher error than the serial offline algorithm at low W/H due to imperfect cache partitioning and eviction decisions, when $W/H \geq 16$, both algorithms converge to the same error. The reason for this convergence is that, at $W/H \geq 16$, both algorithms are forced to delay their flushes until after the close of the current subwindow, as the staleness limit for the previous subwindow has not yet been reached. As a result, both algorithms reduce to pure batching with prioritized flushing. Because the decision of which keys to flush is made after all keys have arrived, there is no prediction error, and it is possible to make the optimal decision of which keys to flush.

This is not to say, however, that this serial approach is optimal overall. The serial approach minimizes the error of each subwindow given the staleness constraint, but as Figure 6.7(c) showed, error is not uniformly distributed across windows. As a result, a minimum-error length-W window does not necessarily comprise minimum-error length-H subwindows.

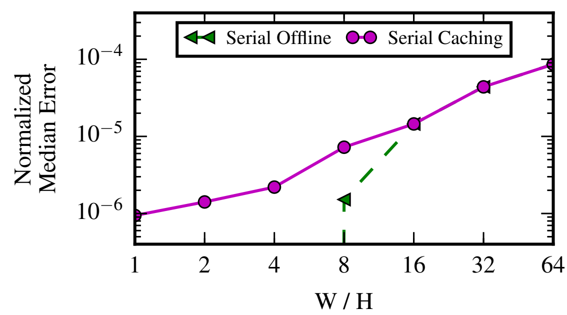
As with the error-bound case, an optimal algorithm therefore needs to make decisions not just across multiple keys within a window, but across multiple windows. Developing such algorithms, and execution-engine support for the associated richer triggering mechanisms are both promising future research directions.



(a) Mean traffic normalized relative to Streaming.



(b) Median staleness normalized by the window length.



(c) Median error normalized by the maximum possible error for our inputs.

Figure 6.9: Traffic, staleness, and error for a `Sum` aggregation under a staleness constraint of $\frac{W}{16}$.

6.5.2 Out-of-Order Arrivals

In practice, records may arrive at the edge out of order with respect to the time they were generated (i.e., their event time). Whether due to intermittent connectivity from data sources to edges, heterogeneous network behavior, or even system failures, systems must gracefully handle these out-of-order arrivals. Out-of-order arrivals significantly complicate matters in practice, primarily because systems can no longer be certain of the completeness of their inputs. Instead, modern systems rely on heuristics to determine the progress of time in their input streams [72].

Beam uses the common abstraction of low *watermarks* to track the progress of event time. The purpose of a watermark is to indicate the oldest event time that could arrive in the future. In practice, watermarks are typically generated using heuristics, and these heuristics are often informed by characteristics of the particular data source. For example, if data originates from log files, file metadata may be used to help estimate the oldest not-yet-ingested data.

Because these watermarks are imperfect, some arrivals will necessarily arrive with event times older than the watermark. How these *late data* are handled depends on the application, based on its answers to Beam's questions of *when* to materialize results and *how* refinements relate.

Throughout the remainder of this chapter, we study the impacts of out-of-order data and associated watermark heuristics on timeliness, accuracy, and cost for grouped aggregation, for both tumbling and hopping windows.

Methodology

To simulate out-of-order arrivals, we preprocess our anonymized Akamai trace to apply a random *delay* to each arriving record. Specifically, we draw the delay for each record from an exponential distribution, and apply this delay to the event time of each record to determine the arrival time for that record. Records are then sorted by their (delayed) arrival time, and our simulator processes arrivals in arrival-time order.

To study the effect of delay, we consider delay distributions with means of 0.1%, 1%, 10%, and 100% of the window length. Because the delays are random, we run each simulation five times; each plotted data point reflects a mean across five runs. For

clarity, error bars are not shown, as most 95% confidence intervals are too narrow to be visible in our plots, and in no case are the confidence intervals wide enough to change any of the conclusions presented here.

In terms of watermarks, we employ a simple and practical heuristic. Our simulator computes the delay for each arrival by subtracting its event time from its arrival time, and it maintains a sliding window of recent delays.[8] We implement *percentile* watermark heuristics which track the specified percentile of this delay, and assume that the oldest arrivals lag behind the newest arrivals by this value. For example, if the 99th-percentile delay is one minute and the newest event time observed by the system is 3:00pm, then the 99th-percentile heuristic watermark is 2:59pm. Using a percentile watermark allows us to roughly control the fraction of arrivals that are deemed late. For example, when using a 99th-percentile watermark, roughly 1% of arrivals will have event times older than the watermark.

Of course it is possible to develop much more sophisticated and accurate watermark heuristics, but that is not our focus. Instead, our goal is to simulate a reasonable and controllable heuristic, and to study how this heuristic affects performance. To that end, we consider 95th-, 99th-, and 99.9^{th}-percentile heuristic watermarks in this chapter.

Algorithms

Given the infeasibility of expressing staleness-constrained algorithms using the current Beam abstractions, we limit our focus to error-constrained algorithms. In particular, we consider three practical online algorithms that can be implemented under the current Beam model, and we consider only the serial-window approach, as this outperforms the parallel-window approach.[9]

Batching: This algorithm captures the default behavior in Beam. Results are materialized (i.e., triggers "fire" in Beam terminology) when the watermark advances past the end of the window. Late arrivals are disregarded, and these omissions are therefore the sole source of error.

[8] We track the most recent 10,000 delays in our simulation, though more sophisticated techniques could achieve similar results with a smaller sliding window.

[9] Recall that the duplicate-elimination technique introduced earlier is only applicable in offline settings.

Batching+LF: This algorithm extends the Batching algorithm by adding *late firings*: late arrivals are immediately streamed to the center. As a result, this approach yields higher traffic but guarantees (eventually) exact results, as all arrivals are ultimately reflected in updates flushed to the center, either as a part of the initial batch for a window, or in the stream of late flushes triggered by late arrivals.

Lazy: This algorithm uses our Lazy triggering policy. Recall that this algorithm tracks the elapsed time necessary for a key to first accumulate enough error to require flushing (t_i) and it schedules the first flush for time $T + W - t_i$, where $T + W$ denotes the end of the window. Additional flushes are triggered if and when the accumulated error again reaches the error tolerance. Note that this algorithm monitors the progress of time and schedules flushes based on event time (i.e., watermark time).

The salient distinction between the Batching algorithms and the Lazy algorithm is that the former base their triggering decisions solely on the progress of event time, whereas the Lazy algorithm considers the actual *values* of arriving data, relying on time as merely one input to its more robust triggering decisions.

We consider a `Sum` aggregation throughout these simulations.

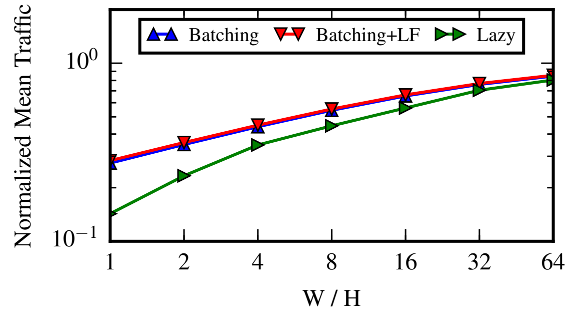
Hopping Windows

We begin by exploring traffic, staleness, and error for these three algorithms as the ratio W/H varies for hopping windows, as shown in Figure 6.10. We consider a mean delay of 1% of the window length, and use a 99th-percentile watermark.

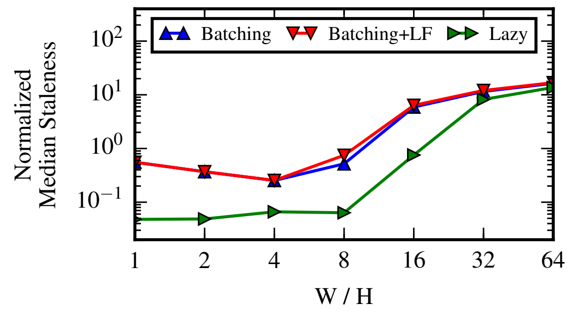
We again see the expected trend: traffic and staleness increase with W/H. We also observe that the value-based Lazy triggering approach yields better traffic, staleness, and error than the baseline time-based approaches.[10] Note that the late-firing variant adds only a small amount of additional traffic over the baseline Batching algorithm, as our heuristic watermark yields only about 1% late data.

The baseline Batching algorithm avoids some flushes by disregarding all late arrivals,

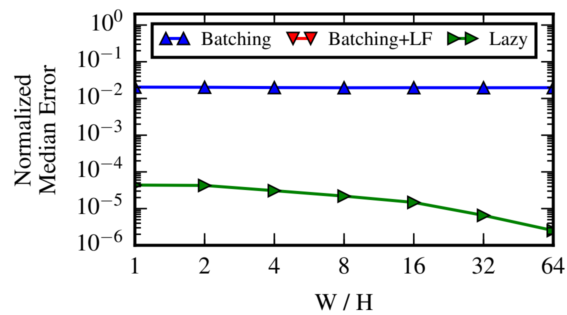
[10] Strictly speaking, Batching+LF achieves *lower* error, but this is not meaningfully *better*, since the application is explicitly error-tolerant.



(a) Mean traffic normalized relative to Streaming.



(b) Median staleness normalized by the window length.



(c) Median error normalized by the maximum possible error for our inputs.

Figure 6.10: Traffic, staleness, and error for a `Sum` aggregation under an error constraint, using a 99$^{\text{th}}$-percentile heuristic watermark, for arrivals with mean delay of 1% of the window length.

while Batching+LF generates an additional update for every late arrival. The value-based Lazy algorithm, on the other hand, only flushes keys that strictly must be flushed to satisfy the error constraint. This more principled approach can significantly reduce traffic, as shown in Figure 6.10(a).

Because the Lazy algorithm reduces overall traffic while also making many flushes in advance of the end of the window, it also achieves much lower staleness than the Batching algorithms. Figure 6.10(b) shows that the staleness impact of late firings is small. Much of the staleness arises from the delay in communicating the burst of traffic at the end of the window as opposed to the additional delay of streaming late arrivals.

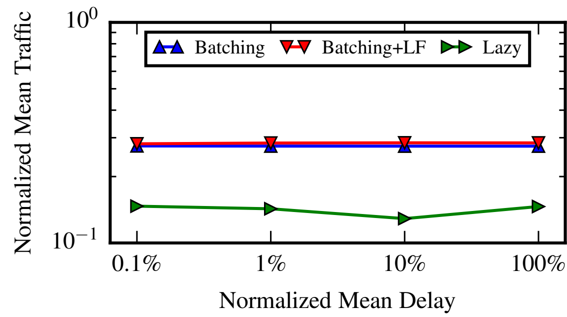
In terms of error, Figure 6.10(c) shows that the advantage of a value-based approach is even more pronounced. The reason is that a value-based approach such as the Lazy algorithm makes systematic decisions about which flushes to avoid, whereas the purely time-based approach is oblivious to the importance of arrivals, in particular late arrivals. Note that the late-firing approach achieves zero error, as all late arrivals are streamed to the center.

Overall, we see that a value-based triggering policy achieves better traffic, staleness, and error than a time-based trigger. For applications that value the timeliness-accuracy-cost tradeoff, it is much better to use a value-based approach. Fortunately, even a simple value-based approach like the Lazy algorithm we use here can be implemented in Beam for an error-bound algorithm. With better algorithms, and extensions to execution engines to support cross-key, cross-window triggering policies, the advantage of value-based approaches over time-based approaches would only widen.

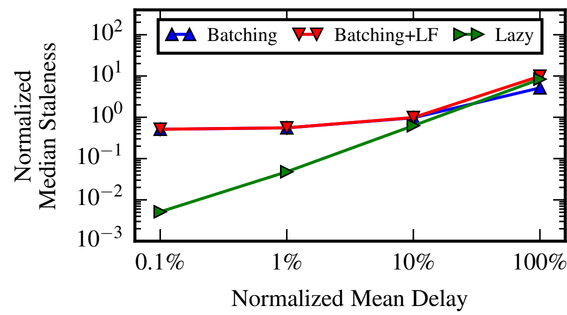
Effect of Delay

To dig deeper, we explore how traffic, staleness, and error vary with the mean delay. Figure 6.11 shows these key metrics for the two time-based Batching algorithms and the value-based Lazy algorithm for tumbling windows with a range of delays applied to our Akamai trace. We see no clear trend in traffic or error as mean delay grows. The reason is that mean delay has no systematic effect on *which* keys are late, or how many keys receive sufficient arrivals to warrant flushing an update.

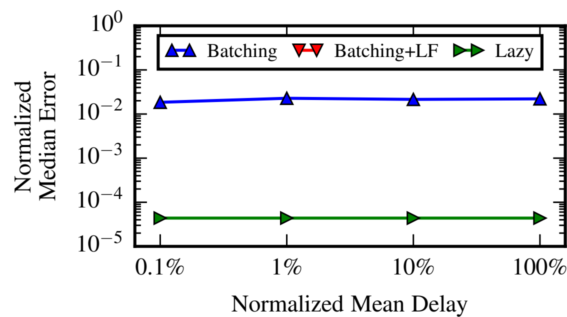
Perhaps the most striking result is that for staleness in Figure 6.11(b). Here we see that staleness for the value-based Lazy algorithm varies closely with mean delay.



(a) Mean traffic normalized relative to Streaming.



(b) Median staleness normalized by the window length.



(c) Median error normalized by the maximum possible error for our inputs.

Figure 6.11: Traffic, staleness, and error for a `Sum` aggregation under an error constraint for tumbling windows, using a 99th-percentile heuristic watermark.

When arrivals are minimally delayed, staleness is also small. The time-based Batching algorithms do not exhibit this same behavior. When delay is low, these algorithms still yield high staleness, as the entire batch of updates needs to be flushed after the end of the window. In other words, when arrivals are minimally delayed, their stalenesses arises from WAN delays. When arrivals are highly delayed, however, the delay of the incoming data begins to overshadow the delay in flushing updates.

At very high delay, we see that the Batching algorithm achieves lower staleness than the Lazy algorithm, but this is only possible because the Batching algorithm neglects several critically important updates, and in doing so yields error more than two orders of magnitude greater than our Lazy algorithm. Aside from this single case, however, we again see that a value-based approach dominates a time-based approach by achieving better traffic, staleness, and error.

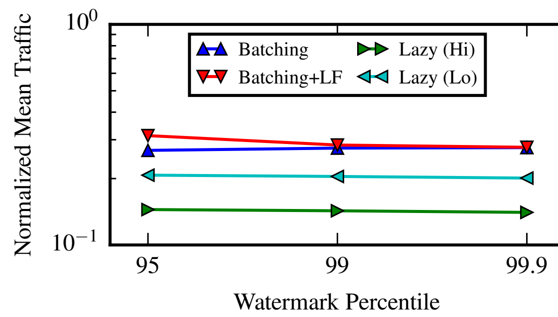
Effect of Watermark Heuristic

Finally, we explore the effect of varying the watermark heuristic, as this is one of the main knobs for trading off timeliness and accuracy in state-of-the-art stream processing systems. Figure 6.12 shows traffic, staleness, and error for the two batching algorithms and our Lazy algorithm. We show two variants of our Lazy algorithm: one with the same error constraint as in the previous figures—Lazy (Hi)—and one with a 50% lower error constraint—Lazy (Lo)—to illustrate a value-based approach's ability to precisely constrain error.

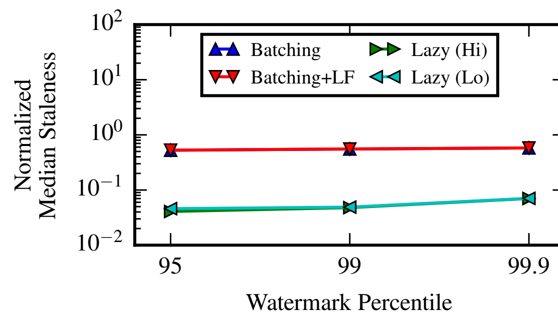
From Figure 6.12(a), we see that the traffic penalty of late firings decreases as the watermark becomes more conservative. We would expect this trend, as a higher watermark percentile yields fewer late arrivals and therefore fewer redundant flushes for the late-firing algorithm.

As the watermark percentile increases, staleness for the Batching algorithms increases very slightly (Figure 6.12(a)). This happens because a stricter, more conservative watermark lags further behind real-time than a more optimistic watermark.

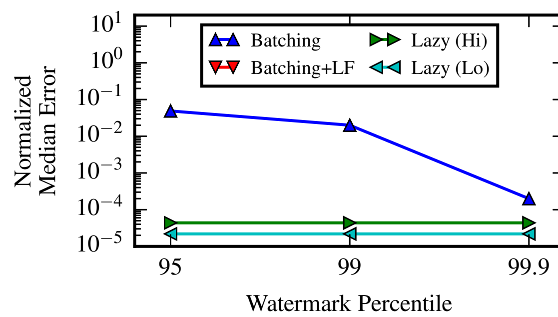
The trend in staleness is more pronounced for the Lazy algorithms, although they yield staleness roughly an order of magnitude lower than the Batching algorithms. Although the Lazy algorithms do not wait for the watermark to advance past the end of the window to *begin* flushing updates, they do base their scheduling decisions on the



(a) Mean traffic normalized relative to Streaming.



(b) Median staleness normalized by the window length.



(c) Median error normalized by the maximum possible error for our inputs.

Figure 6.12: Traffic, staleness, and error for a `Sum` aggregation under an error constraint for tumbling windows, for arrivals with mean delay of 1% of the window length.

watermark progress. As a result, a more conservative watermark will lead to slightly later flushes and in turn slightly higher staleness for the Lazy algorithms.

Figure 6.12(c) demonstrates the most pronounced effect of the watermark heuristic. For the Batching algorithm, a more conservative watermark (i.e., a higher watermark percentile) leads to fewer arrivals being considered late, and in turn lower error. As a result, the watermark heuristic can have a significant effect on error for a time-based triggering policy. It is worth noting that this figure shows median error over sixty windows; the trend in maximum error is much less pronounced, illustrating that, while the watermark heuristic can affect error, it does not afford the same precise control as we see from a value-based triggering policy.

Overall, it is much more powerful to use a value-based triggering policy along the lines of the Lazy algorithm. For one, value-based algorithms allow us to *explicitly* control the error, rather than requiring a trial-and-error process as would be required for the time-based approaches. By comparing Lazy (Lo) with Lazy (Hi), we can see how a value-based algorithm can deliver an arbitrary error constraint while making systematic tradeoffs for the other metrics of staleness and traffic. Second, value-based policies can achieve lower traffic, staleness, and error than the time-based approaches, as we have consistently observed in this section.

Again, it is worth emphasizing that the Lazy algorithm considered here, while inspired by the offline optimal error-constrained algorithm from Chapter 5, could be improved upon by allowing cross-key and network-aware triggering decisions. Further, we know that the serial-window approach is suboptimal, so it is possible to devise fundamentally more effective algorithms.

Such algorithmic and engine-level improvements would only strengthen the advantage of principled value-based triggering policies over the current state of the art.

6.6 Concluding Remarks

While stream and batch computing have historically been viewed separately, it is increasingly important to develop both programming models and execution engines that unify them. Progress toward a unified programming model has been rapid, and new

engines are constantly evolving. The current state of the art, however, requires programmers to answer the low-level question of when to materialize results. We argue that application developers are more concerned with their high-level requirements in terms of timeliness, accuracy, and cost, and that the choice of when to materialize results is a complex one that must consider dynamic network and data characteristics. Systems, therefore, must accept the responsibility for dynamically determining when to materialize results.

There are numerous research challenges to be addressed both for programming models and execution engines. Models should be extended to allow application developers to concisely and explicitly express their requirements along the dimensions of timeliness, accuracy, and cost. Execution engines must evolve to support more sophisticated triggering policies that can make decisions not just for individual keys in isolation, but also across multiple keys and windows, and with better runtime information such as awareness of network congestion.

Meanwhile, the optimization techniques employed by execution engines should be extended to address the new challenges that arise from more general windowing strategies and out-of-order inputs. The techniques presented in this thesis provide useful starting points.

Chapter 7

Conclusion

Many modern applications need to process a large volume of geographically distributed data on a geographically distributed platform with low latency. One key challenge in achieving this requirement is determining *where* to carry out the computation.

For applications that process unbounded *streams* of geo-distributed data, two key performance metrics are critical: *WAN traffic* and *staleness*. To achieve the goals of low traffic and low staleness, a system must determine *when to communicate results* from the edges to the center.

As an additional challenge, constrained WAN bandwidth often renders exact computation with bounded staleness infeasible. Many real-world applications can tolerate some staleness or inaccuracy in their final results, albeit with diverse preferences. In order to support these diverse preferences and achieve the desired staleness-error trade-off, geo-distributed stream processing systems must determine *what partial results to communicate*.

In this thesis, we presented systematic answers to these three critical questions—where to compute, when to communicate, and what partial results to communicate—in both the batch and stream computing contexts. We also demonstrated the potential for these techniques and outlined open questions in the context of emerging systems for unified stream and batch computing.

7.1 Research Contributions

7.1.1 Batch Processing

In the batch processing context, we focused on the question of *where* to perform computation, specifically for MapReduce applications. We developed model-driven optimization techniques and applied the resulting insights to develop practical techniques for a real-world MapReduce implementation.

Model-Driven Optimization. We devised a model to predict the execution time of MapReduce applications in geographically distributed environments and an optimization framework for determining the best placement of data and computation. We used our model and optimization to derive several insights, for example that application characteristics can significantly influence the optimal data and computation placement, and that our optimization yields greater benefits as the environment becomes increasingly distributed. Further, we experimentally demonstrated that our optimization approach can significantly outperform standard Hadoop.

Practical Systems Techniques. Based on the insights from our model-driven optimization, we developed practical techniques for real-world MapReduce implementations such as Hadoop. Our *Map-Aware Push* technique optimizes the push and map phases by enabling push-map overlap that allows map behavior to feed back and affect the push phase. Similarly, our *Shuffle-Aware Map* technique optimizes the map and reduce phases to enable shuffle cost to feed back and affect map scheduling decisions. We demonstrated the effectiveness of these techniques through experiments on geographically distributed testbeds on both Amazon EC2 and PlanetLab.

7.1.2 Stream Processing

In the stream processing context, we developed techniques for jointly optimizing timeliness and cost for exact computation, and for trading off timeliness and accuracy for approximate computation.

Exact Computation. Our first contribution in the streaming context focused on applications requiring exact results. We examined the question of when to communicate results from the edges to the center in the context of *windowed grouped aggregation* in order to optimize two key metrics of any geo-distributed streaming analytics service: *WAN traffic* and *staleness*. We presented a family of optimal offline algorithms that *jointly minimize both staleness and traffic*. Building on this foundation, we developed practical online aggregation algorithms based on the observation that grouped aggregation can be modeled as a *caching* problem where the cache size varies over time.

We demonstrated the practicality of these algorithms through an implementation in Apache Storm, deployed on a PlanetLab testbed. Our experiments, driven by workloads derived from anonymized traces of a web analytics service offered by a large commercial CDN, showed that our online aggregation algorithms perform close to the optimal offline algorithms for a variety of system configurations, stream arrival rates, and query types.

Approximate Computation. Continuing our focus on windowed grouped aggregation, we presented optimal offline algorithms for minimizing staleness under an error constraint and for minimizing error under a staleness constraint. Using these offline algorithms as references, we developed practical online algorithms for effectively trading timeliness and accuracy under bandwidth limitations. We evaluated these techniques through both simulation and experiments, and showed that our algorithms significantly reduce staleness and error relative to a practical random sampling/batching-based baseline across a diverse set of aggregation functions.

7.1.3 Unified Stream & Batch Processing

Finally, we focused on the current trend toward a unified programming model capable of expressing both batch and stream programs [15, 16] and unified computing engines capable of efficiently computing over both bounded batches and unbounded streams [17, 18]. We argued that the state-of-the-art programming model is both burdensome and insufficiently expressive for applications that value performance along the key dimensions of timeliness, accuracy, and cost. We also highlighted opportunities to apply our optimization techniques to emerging execution engines, and noted where these techniques need to be extended.

7.2 Future Research Directions

While the work presented here advances the state of the art, it also illuminates open problems worthy of future research. Here we briefly describe several possible research directions.

7.2.1 Alternative Performance Metrics and Optimization Objectives

Throughout this thesis, we have focused on performance along the key dimensions of timeliness, accuracy, and cost. Specifically, we minimized makespan in the batch computing context, and focused on staleness, error, and WAN traffic in the streaming context. While these are critical metrics for many applications and systems, they are certainly not the only metrics worth optimizing.

Storage and energy are two other dimensions to consider. For example, storage constraints—which may exhibit pronounced heterogeneity in geo-distributed settings—can restrict the options for *where* to place computation. Further, a stream processing system may be forced to communicate partial results early if it lacks sufficient storage. The techniques throughout this thesis could be extended to incorporate storage constraints.

Minimizing energy consumption (both average and peak) is increasingly important. Energy consumption for an analytics application is a non-trivial function of its storage, communication, and computation requirements, all of which vary based on query and data characteristics. Our model-driven optimization for geo-distributed MapReduce could be extended to incorporate energy constraints. An alternative formulation could use energy consumption as an optimization objective subject to a constraint on makespan. In the stream computing context, we have assumed that staleness arises due to constrained WAN bandwidth. Under an energy constraint, or with an objective of minimizing energy consumption, however, the bottleneck may lie elsewhere. Studying the tradeoffs between energy consumption and other key performance metrics would be an interesting future direction.

Even within the timeliness-accuracy-cost tradeoff space we have studied here, there are alternative problem formulations that may be more appropriate for some applications. For example, in a deployment with limited compute capacity at the edges, it

may be necessary to apply (or develop new) load-shedding techniques, or to focus on architectures other than the hub-and-spoke model we have considered (e.g., hierarchical topologies). In other cases, it is possible that WAN bandwidth, while costly, is constrained at a system level, but plentiful relative to any individual application. In such cases, it may be worthwhile to extend our caching-based techniques to operate under a bandwidth constraint, or at a higher level to develop techniques for optimizing the allocation of bandwidth across multiple applications. It may also be interesting to incorporate compute and storage costs, and to study scenarios where costs vary with time, for example due to fluctuating demand.

7.2.2 Alternative Applications

This thesis has focused primarily on applications that can be implemented via grouped aggregation, and in the streaming context we have largely confined our attention to tumbling windows. As Chapter 6 highlighted, even a seemingly minor generalization—in particular, hopping windows—can uncover a host of interesting research challenges. Session windows [15] introduce yet more challenges, and would be an interesting focal point for further research.

We have considered computation over geographically distributed data, but we have not yet studied geography as an *attribute* of the data. In many compelling real-world applications, however, geography is a critical data attribute [96]. For example, an analytics user might perform grouped aggregation to track the popularity of live video streams by city or state, to understand the effectiveness of an advertising campaign in different regions, or to understand trending conversation topics in different countries. In many cases, the geographic attributes of data may correlate with the physical location of data sources, and there may be interesting opportunities to leverage this correlation to inform the decision of where to place computation.

Moving beyond aggregation would expose further research challenges. A particularly interesting topic would be iterative applications. We have focused throughout this thesis on applications where data flows through the system only once, but many modern applications, especially those over graphs [97, 98], process inputs repeatedly until converging to a final result. This pattern of iterative refinement exposes an interesting opportunity for trading timeliness for accuracy and cost. In addition, systems

for iterative computation have typically been confined to a single datacenter, and with relatively few exceptions (e.g., Kineograph [99], Naiad [100]), they have assumed static batch inputs. Optimizing timeliness, accuracy, and cost for iterative applications in geographically distributed settings—and additionally over streaming inputs—would yield a host of fascinating new research challenges.

7.2.3 Scalable System Implementations

We have focused more in this thesis on general application models (MapReduce and windowed grouped aggregation) than on specific implementations (e.g., Hadoop, Flink). While we have experimentally demonstrated that our techniques can apply in real-world systems, scaling our techniques—both across resources and across applications—presents several interesting challenges. As discussed in Section 5.6, coordination across multiple edges could improve our ability to trade timeliness and accuracy in geo-distributed stream computing. Distributed function monitoring techniques [61, 88] may be a useful starting point. Providing richer triggering abstractions in Beam, while scaling execution engines across multiple cores, nodes, and edges, is also an important direction, as discussed in Chapter 6.

Scaling to multiple applications is another interesting direction. For example, Clarinet [101] considers collections of batch jobs in geo-distributed settings, and selects query execution *plans* that share common sub-computations and *schedules* that consider the interactions between jobs. Meanwhile, Vega [102] focuses on batch jobs in a single datacenter and reuses relevant existing results when queries are modified and rerun. These systems both make their primary contributions at the level of query planning and optimization. If these techniques were extended to address geo-distributed streaming analytics, then we could use them to optimize query plans while using the techniques in this thesis to answer the lower-level questions of where to place computation, when to communicate partial results, and what partial results to communicate. As we discussed in Section 2.4.3, there may be interesting interactions between these static and dynamic optimization techniques that warrant further research.

References

[1] Q4 15 letter to shareholders, 2016. http://ir.netflix.com.

[2] Evolution of the netflix data pipeline, 2016. http://techblog.netflix.com/2016/02/evolution-of-netflix-data-pipeline.html.

[3] Facebook reports fourth quarter and full year 2015 results, 2016. http://investor.fb.com/releasedetail.cfm?ReleaseID=952040.

[4] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. An analysis of facebook photo caching. In *Proc. of SOSP*, pages 167–181, November 2013.

[5] Teaching machines to see and understand, 2015. https://research.facebook.com/blog/teaching-machines-to-see-and-understand.

[6] Computing, 2015. http://home.cern/about/computing.

[7] Detection — LIGO lab — Caltech, 2016. https://www.ligo.caltech.edu/detection.

[8] Ashish Vulimiri, Carlo Curino, Brighten Godfrey, Konstantinos Karanasos, and George Varghese. WANalytics: Analytics for a geo-distributed data-intensive world. In *Proc. of CIDR*, January 2015.

[9] Erik Nygren, Ramesh K. Sitaraman, and Jennifer Sun. The Akamai network: a platform for high-performance internet applications. *SIGOPS Oper. Syst. Rev.*, 44(3):2–19, August 2010.

[10] Tony Hey, Stewart Tansley, and Kristin Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery.* Microsoft Research, Redmond, Washington, 2009.

[11] A conversation with Jim Gray. *Queue*, 1(4):8–17, June 2003.

[12] Ariel Rabkin, Matvey Arye, Siddhartha Sen, Vivek S. Pai, and Michael J. Freedman. Aggregation and degradation in JetStream: Streaming analytics in the wide area. In *Proc. of NSDI*, pages 275–288, 2014.

[13] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. of OSDI*, pages 137–150, 2004.

[14] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. PlanetLab: an overlay testbed for broad-coverage services. *ACM SIGCOMM*, 33(3):3–12, 2003.

[15] Tyler Akidau et al. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. of the VLDB Endowment*, 8:1792–1803, 2015.

[16] Apache beam (incubating), June 2016. `http://beam.incubator.apache.org/`.

[17] Apache Flink: Scalable Batch and Stream Data Processing, 2016. `https://flink.apache.org/`.

[18] Apache Apex, 2016. `https://apex.apache.org/`.

[19] Benjamin Heintz, Chenyu Wang, Abhishek Chandra, and Jon Weissman. Cross-phase optimization in MapReduce. In *Proc. of IEEE IC2E*, pages 338–347, March 2013.

[20] Benjamin Heintz, Abhishek Chandra, Ramesh K. Sitaraman, and Jon Weissman. End-to-end optimization for geo-distributed MapReduce. *IEEE Transactions on Cloud Computing*, 4(3):293–306, July 2016.

[21] Benjamin Heintz, Abhishek Chandra, and Ramesh K. Sitaraman. Towards optimizing wide-area streaming analytics. In *Proc. of the 2nd IEEE Workshop on Cloud Analytics*, 2015.

[22] Benjamin Heintz, Abhishek Chandra, and Ramesh K. Sitaraman. Optimizing grouped aggregation in geo-distributed streaming analytics. In *Proc. of HPDC*, pages 133–144, June 2015.

[23] Benjamin Heintz, Abhishek Chandra, and Ramesh K. Sitaraman. Trading timeliness and accuracy in geo-distributed streaming analytics. In *Proc. of ACM SoCC*, pages 361–373, 2016.

[24] Hadoop, 2016. `http://hadoop.apache.org`.

[25] Sriram Rao, Raghu Ramakrishnan, Adam Silberstein, Mike Ovsiannikov, and Damian Reeves. Sailfish: A framework for large scale data processing. In *Proc. of ACM SoCC*, pages 4:1–4:14, 2012.

[26] Herodotos Herodotou, Fei Dong, and Shivnath Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *Proc. of ACM SoCC*, pages 18:1–18:14, 2011.

[27] Virajith Jalaparti, Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Antony Rowstron. Bridging the Tenant-Provider Gap in Cloud Services. In *Proc. of ACM SoCC*, pages 6:1–6:14, 2012.

[28] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce Online. In *Proc. of NSDI*, pages 313–327, 2010.

[29] Abhishek Verma, Nicolas Zea, Brian Cho, Indranil Gupta, and Roy H. Campbell. Breaking the MapReduce stage barrier. In *Proc. of IEEE Cluster*, pages 235–244, 2010.

[30] H.P. Williams. *Model building in mathematical programming*. Wiley, 1999.

[31] Free eBooks by Project Gutenberg. `http://www.gutenberg.org/`.

[32] Martin Arlitt and Tai Jin. Workload characterization of the 1998 World Cup Web Site. Technical Report HPL-1999-35R1, HP Labs, September 1999.

[33] Jimmy Lin and Chris Dyer. *Data-Intensive Text Processing with MapReduce.* Morgan and Claypool Publishers, 2010.

[34] Steve Ko, Imranul Hoque, Brian Cho, and Indranil Gupta. Making Cloud Intermediate Data Fault-Tolerant. In *Proc. of ACM SoCC*, pages 181–192, 2010.

[35] Balaji Palanisamy, Aameek Singh, Ling Liu, and Bhushan Jain. Purlieus: locality-aware resource allocation for MapReduce in a cloud. In *Proc. of ACM SC*, pages 58:1–58:11, 2011.

[36] Mohammad Hammoud, M. Suhail Rehman, and Majd F. Sakr. Center-of-Gravity reduce task scheduling to lower MapReduce network traffic. In *Proc. of IEEE Cloud*, pages 49–58, 2012.

[37] Hrishikesh Gadre, Ivan Rodero, and Manish Parashar. Investigating MapReduce framework extensions for efficient processing of geographically scattered datasets. *SIGMETRICS Perf. Eval. Rev.*, 39(3):116–118, 2011.

[38] Shingyu Kim, Junghee Won, Hyuck Han, Hyeonsang Eom, and Heon Y. Yeom. Improving Hadoop performance in intercloud environments. In *Proc. of ACM SIGMETRICS*, pages 107–109, 2011.

[39] Michael Cardosa, Chenyu Wang, Anshuman Nangia, Abhishek Chandra, and Jon Weissman. Exploring MapReduce Efficiency with Highly-Distributed Data. In *Proc. of MAPREDUCE*, pages 27–33, 2011.

[40] Yuan Luo, Zhenhua Guo, Yiming Sun, Beth Plale, Judy Qiu, and Wilfred W. Li. A hierarchical framework for cross-domain MapReduce execution. In *Proc. of ECMLS*, pages 15–22, 2011.

[41] Matei Zaharia, Andrew Konwinski, Anthony D. Joseph, Randy H. Katz, and Ion Stoica. Improving MapReduce performance in heterogeneous environments. In *Proc. of OSDI*, pages 29–42, 2008.

[42] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, and Bikas Saha. Reining in the outliers in map-reduce clusters using mantri. In *Proc. of OSDI*, pages 265–278, 2010.

[43] Faraz Ahmad, Srimat Chakradhar, Anand Raghunathan, and T. N. Vijaykumar. Tarazu: Optimizing MapReduce on heterogeneous clusters. In *Proc. of ASPLOS*, pages 61–74, 2012.

[44] Michael Cardosa, Piyush Narang, Abhishek Chandra, Himabindu Pucha, and Aameek Singh. STEAMEngine: Driving MapReduce Provisioning in the Cloud. In *Proc. of HiPC*, 2011.

[45] 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 *Proc. of CIDR*, pages 261–272, 2011.

[46] Thomas Sandholm and Kevin Lai. MapReduce optimization using dynamic regulated prioritization. In *Proc. of ACM SIGMETRICS*, pages 299–310, 2009.

[47] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proc. of EuroSys*, pages 265–278, 2010.

[48] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proc. of SOSP*, pages 261–276, 2009.

[49] Rich Wolski, Neil T. Spring, and Jim Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computing Systems*, 15:757–768, 1999.

[50] Jinoh Kim, Abhishek Chandra, and Jon Weissman. Passive network performance estimation for large-scale, data-intensive computing. *IEEE TPDS*, 22(8):1365–1373, Aug. 2011.

[51] Qi He, Constantine Dovrolis, and Mostafa Ammar. On the predictability of large transfer TCP throughput. In *Proc. of ACM SIGCOMM*, pages 145–156, 2005.

[52] Quan Chen, Daqiang Zhang, Minyi Guo, Qianni Deng, Song Guo, and Song Guo. SAMR: A self-adaptive MapReduce scheduling algorithm in heterogeneous environment. In *Proc. of IEEE CIT*, pages 2736–2743, 2010.

[53] Heshan Lin, Xiaosong Ma, Jeremy Archuleta, Wu-chun Feng, Mark Gardner, and Zhe Zhang. MOON: MapReduce on opportunistic environments. In *Proc. of HPDC*, pages 95–106, 2010.

[54] Fernando Costa, Luis Silva, and Michael Dahlin. Volunteer cloud computing: MapReduce over the internet. In *Proc. of IEEE IPDPSW*, pages 1855–1862, 2011.

[55] Shivnath Babu. Towards automatic optimization of MapReduce programs. In *Proc. of ACM SoCC*, pages 137–142, 2010.

[56] GridFTP. `http://globus.org/toolkit/docs/3.2/gridftp/`.

[57] BitTorrent. `http://www.bittorrent.com`.

[58] Storm, distributed and fault-tolerant realtime computation. `http://storm.apache.org/`, 2015.

[59] Oscar Boykin, Sam Ritchie, Ian O'Connel, and Jimmy Lin. Summingbird: A framework for integrating batch and online mapreduce computations. In *Proc. of VLDB*, volume 7, pages 1441–1451, 2014.

[60] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proc. of SOSP*, pages 423–438, 2013.

[61] Graham Cormode. Continuous distributed monitoring: A short survey. In *Proc. of AlMoDEP*, pages 1–10, 2011.

[62] Stefan Podlipnig and Laszlo Böszörmenyi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, 35(4):374–398, December 2003.

[63] J. Bolliger and T.R. Gross. Bandwidth monitoring for network-aware applications. In *Proc. of HPDC*, pages 241–251, 2001.

[64] google/snappy: A fast compressor/decompressor. `https://github.com/google/snappy/`, 2016.

[65] Philippe Flajolet, Éric Fusy, Olivier Gandouet, et al. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *Proc. of AOFA*, 2007.

[66] Jim Gray et al. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Min. Knowl. Discov.*, 1(1):29–53, January 1997.

[67] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *Proc. of SOSP*, pages 247–260, 2009.

[68] P.-A. Larson. Data reduction by partial preaggregation. In *Proc. of ICDE*, pages 706–715, 2002.

[69] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: A Tiny AGgregation service for ad-hoc sensor networks. In *Proc. of OSDI*, 2002.

[70] R. Rajagopalan and P.K. Varshney. Data-aggregation techniques in sensor networks: A survey. *IEEE Communications Surveys Tutorials*, 8(4):48–63, 2006.

[71] Hrishikesh Amur et al. Memory-efficient groupby-aggregate using compressed buffer trees. In *Proc. of SoCC*, 2013.

[72] Tyler Akidau et al. MillWheel: Fault-tolerant stream processing at internet scale. *Proc. of VLDB Endow.*, 6(11):1033–1044, August 2013.

[73] Sirish Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. of CIDR*, 2003.

[74] Sanjeev Kulkarni et al. Twitter heron: Stream processing at scale. In *Proc. of SIGMOD*, pages 239–250, 2015.

[75] Zhengping Qian et al. TimeStream: reliable stream computation in the cloud. In *Proc. of EuroSys*, pages 1–14, 2013.

[76] Daniel J. Abadi et al. The design of the borealis stream processing engine. In *Proc. of CIDR*, pages 277–289, 2005.

[77] Arvind Arasu et al. STREAM: the Stanford data stream management system. In *Data Stream Management - Processing High-Speed Data Streams*, pages 317–336. 2016.

[78] Qifan Pu et al. Low latency geo-distributed data analytics. In *Proc. of SIGCOMM*, pages 421–434, August 2015.

[79] P. Pietzuch et al. Network-aware operator placement for stream-processing systems. In *Proc. of ICDE*, 2006.

[80] Jeong-Hyon Hwang, U. Cetintemel, and S. Zdonik. Fast and highly-available stream processing over wide area networks. In *Proc. of ICDE*, pages 804–813, 2008.

[81] James Cipar et al. LazyBase: trading freshness for performance in a scalable database. In *Proc. of EuroSys*, pages 169–182, 2012.

[82] Sameer Agarwal et al. BlinkDB: queries with bounded errors and bounded response times on very large data. In *Proc. of EuroSys*, pages 29–42, 2013.

[83] Tathagata Das, Yuan Zhong, Ion Stoica, and Scott Shenker. Adaptive stream processing using dynamic batch sizing. In *Proc. of SoCC*, pages 16:1–16:13, 2014.

[84] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A blueprint for introducing disruptive technology into the internet. *SIGCOMM Comput. Commun. Rev.*, 33(1):59–64, January 2003.

[85] Graham Cormode, Theodore Johnson, Flip Korn, S. Muthukrishnan, Oliver Spatscheck, and Divesh Srivastava. Holistic UDAFs at streaming speeds. In *Proc. of SIGMOD*, pages 35–46, 2004.

[86] Graham Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, April 2005.

[87] Bradley Efron. Better bootstrap confidence intervals. *Journal of the American Statistical Association*, 82(397):171–185, 1987.

[88] Nikos Giatrakos, Antonios Deligiannakis, and Minos Garofalakis. Scalable approximate query tracking over highly distributed data streams. In *Proc. of SIGMOD*, pages 1497–1512, 2016.

[89] Mark Boddy. Anytime problem solving using dynamic programming. In *Proc. of AAAI*, pages 738–743, 1991.

[90] Shlomo Zilberstein. Operational rationality through compilation of anytime algorithms. Technical report, 1993.

[91] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proc. of SOSP*, pages 29–43, 2003.

[92] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proc. of MSST*, pages 1–10, 2010.

[93] Trident API Overview. `http://storm.apache.org/releases/current/Trident-API-Overview.html`.

[94] Nathan Marz and James Warren. *Big Data: Principles and best practices of scalable real-time data systems*. Manning, April 2015.

[95] Questioning the Lambda Architecture. `https://www.oreilly.com/ideas/questioning-the-lambda-architecture`.

[96] A. Magdy, M. F. Mokbel, S. Elnikety, S. Nath, and Y. He. Venus: Scalable real-time spatial queries on microblogs with adaptive load shedding. *IEEE Transactions on Knowledge and Data Engineering*, 28(2):356–370, Feb 2016.

[97] Grzegorz Malewicz et al. Pregel: A system for large-scale graph processing. In *Proc. of SIGMOD*, pages 135–146, 2010.

[98] Joseph E. Gonzalez et al. GraphX: Graph processing in a distributed dataflow framework. In *Proc. of OSDI*, pages 599–613, 2014.

[99] Raymond Cheng et al. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proc. of EuroSys*, pages 85–98, 2012.

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

[101] Raajay Viswanathan, Ganesh Ananthanarayanan, and Aditya Akella. CLARINET: WAN-aware optimization for analytics queries. In *Proc. of OSDI*, pages 435–450, November 2016.

[102] Matteo Interlandi et al. Optimizing interactive development of data-intensive applications. In *Proc. of SoCC*, pages 510–522, 2016.