A Platform for Scalable Low-Latency Analytics using MapReduce

Boduo Li
University of Massachusetts - Amherst

A PLATFORM FOR SCALABLE LOW-LATENCY ANALYTICS USING MAPREDUCE

A Dissertation Presented

by

BODUO LI

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2015

Computer Science

A PLATFORM FOR SCALABLE LOW-LATENCY ANALYTICS USING MAPREDUCE

A Dissertation Presented

by

BODUO LI

ACKNOWLEDGMENTS

Foremost, I would like to express my sincere gratitude to my PhD advisor, Prof. Yanlei Diao, without whose guidance and support for years, the thesis would not be possible. I learnt from her not only kills to conduct research, from literature survey and solving technical problems to academic writing and presentation, but also time management, ways of critical thinking, insistence on quality of work and positive attitude towards difficulties. They will be highly valuable to me in my entire career.

I am grateful to all my thesis committee members. I am deeply indebted to Prof. Prashant Shenoy, who co-advised me throughout my thesis work. I benefited a lot from his patience, vision and immense knowledge. With his help, I was able to complete the thesis with the integration of insights across various research areas, such as queuing theory and operating systems. Next, I would like to thank Prof. Andrew McGregor, who I have closely worked with. His knowledge and enthusiasm in algorithm research inspired me to design various algorithms in the thesis. I am also grateful for having Prof. David Irwin on my thesis committee and his comments on the work.

I am grateful to many other professors at UMass. I specially thank Prof. Michael Zink for his patient help in my first year of PhD on a case study that was in a totally unknown research domain to me. I am grateful to Prof. Gerome Miklau and Alexandra Meliou for input and discussions about many aspects of database research.

I would like to thank my colleagues and friends at UMass. I particularly thank Edward Mazur for his close collaboration and his friendship. He performed extensive benchmarks over the existing related systems, such as Hadoop and HOP. I also deeply felt inspired from his devotion to techniques. During my early years at UMass, I collaborated with senior students Thanh Tran, Zhao Cao and Devesh Agrawal. I appreciate that I could have the chance to learn from their experiences, which helped me start my PhD journey more easily. I am grateful to Yue Wang, Chao Li and Bo Jiang for active discussion with me on my research problems, as well as all the members of the database group for sharing thoughts without reservation and making the lab a nice place to work. I thank Rachel Lavery and Leeanne Leclerc for making my PhD study much easier by taking care of administrative issues. I thank Tyler Trafford for maintaining the cluster used in my thesis work. And I would like to thank all my friends at UMass who made my life at Amherst enjoyable and memorable.

During my PhD study, I had an opportunity to work as a summer intern at IBM Almaden Research Center. I would like to thank Dr. Sandeep Tata and Dr. Yannis Sismanis for their great mentorship and the experience of working in an industrial research lab. I am also grateful to all the friends I made there for many laughters in the enjoyable summer.

I am deeply obliged to my parents who are always considerate and supportive when I was facing difficulties during my PhD study. Although I have not been home for a few years, I can always feel their love, which has encouraged me to keep moving forward. I feel so lucky and proud to have them as my parents.

Finally, I want to express my heartfelt thanks to my dear wife, Liping. Since twelve years ago, she accompanied me in every step in my life. She has been my classmate, labmate, and soulmate. She was always there standing by me through the times good and bad. Without her support, the doctorate would not be possible. I dedicate this thesis to her.

ABSTRACT

A PLATFORM FOR SCALABLE LOW-LATENCY ANALYTICS USING MAPREDUCE

MAY 2015

BODUO LI

Bachelor, HARBIN INSTITUTE OF TECHNOLOGY, CHINA

Master, HARBIN INSTITUTE OF TECHNOLOGY, CHINA

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Yanlei Diao

Today, the ability to process "big data" has become crucial to the information needs of many enterprise businesses, scientific applications, and governments. Recently, there have been increasing needs of processing data that is not only "big" but also "fast". Here "fast data" refers to high-speed real-time and near real-time data streams, such as Twitter feeds, search query streams, click streams, impressions, and system logs. To handle both historical data and real-time data, many companies have to maintain multiple systems. However, recent real-world case studies show that maintaining multiple systems cause not only code duplication, but also intensive manual work to partition the analytics workloads and determine which data is processed by which system. These issues point to the need for a general, unified data processing framework to support analytical queries with different latency requirements.

This thesis takes a further step towards building a general, unified system for big and fast data analytics. In order to build such a system, I propose to build on existing solutions on data parallelism and extend them with two new features: *incremental processing* and *stream processing with latency constraints*. This thesis starts with Hadoop, the most popular open-source MapReduce implementation, which provides proven scalability based on data parallelism. I answer the following questions: (1) Is Hadoop able to support incremental processing? (2) What are the necessary architecture changes in order to support incremental processing? (3) What are the additional design features needed to support stream processing with latency constraints? The thesis includes three parts that answer each of the questions.

The first part of the thesis validates whether the existing MapReduce implementations can support incremental processing. Incremental processing means that computation is performed as soon as the relevant data becomes available. My extensive benchmark study of Hadoop-based MapReduce systems shows that the widely-used sort-merge implementation for partitioning and parallel processing poses a fundamental barrier to incremental computation. I further propose a cost model, and optimize the Hadoop system configuration based on the model. The benchmark results over the optimized system verify that the barrier to incremental computation is intrinsic, and cannot be removed by tuning system parameters.

In the second part of the thesis, I employ various purely hash-based techniques to enable fast in-memory incremental processing in MapReduce, and frequent key based techniques to extend such processing to workloads that require memory more than available. I evaluate my Hadoop-based prototype equipped with all proposed techniques. The results show that the hash techniques allow the reduce progress to keep up with the map progress with up to 3 orders of magnitude reduction of internal disk spills, and enable results to be returned early.

The third part of the thesis aims to support stream processing with latency constraints based on the incremental processing platform resulted from the second part. I perform a benchmark study to understand the sources of latency. I then propose a number of necessary architecture changes to support stream processing, and augment the platform with new latency-aware model-driven resource planning and latency-aware runtime scheduling techniques to meet user-specified latency constraints while maximizing throughput. Experiments using real-world workloads show that the techniques reduce the latency from tens or hundreds of seconds to sub-second, with 2x-5x increase in throughput. The new platform offers 1-2 orders of magnitude improvements over Storm, a commercial-grade distributed stream system, and Spark Streaming, a state-of-the-art academic prototype, when considering both latency and throughput.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS . . . . **iv**

ABSTRACT . . . . **vi**

LIST OF TABLES . . . . **xiii**

LIST OF FIGURES . . . . **xv**

CHAPTER

1. INTRODUCTION **1**

1.1 Thesis Contributions 5

1.1.1 Hadoop Benchmarking and Optimization 5
1.1.2 Incremental Processing 6
1.1.3 Stream Processing with Latency Constraints 7
1.1.4 System Development and Code Release 10

1.2 Thesis Organization 11

2. RELATED WORK **12**

2.1 A Survey of Data-Parallel Systems 12

2.1.1 System Design Features for Low-Latency Analytics 12
2.1.2 Comparison of Data-Parallel Systems 14

2.2 Query Model and Query Language 19
2.3 Out-of-order Processing. 20
2.4 Fault-tolerance. 21
2.5 Elasticity 22
2.6 MapReduce 23
2.7 Stream Database Systems 24
2.8 Parallel Database Systems 24
2.9 Other Systems 25

3. HADOOP BENCHMARKING AND OPTIMIZATION 26

3.1 Background 27

3.1.1 Overview of MapReduce 27
3.1.2 Overview of the Hadoop Implementation 28

3.2 Benchmarking and Analysis 30

3.2.1 Experimental Setup 31
3.2.2 Result Analysis 32
3.2.3 Architectural Improvements 37
3.2.4 MapReduce Online 38
3.2.5 Summary of Results 40

3.3 Optimizing Hadoop 41

3.3.1 An Analytical Model for Hadoop 41

3.3.1.1 Modeling I/O Cost in Bytes 42
3.3.1.2 Modeling the Number of I/O requests 45
3.3.1.3 Modeling the Startup Cost 47
3.3.1.4 Combining All in Time Measurement 48

3.3.2 Optimizing Hadoop based on our Analytical Model 48

3.3.2.1 Optimizations 48
3.3.2.2 Analysis of Optimized Hadoop 52

3.3.3 Pipelining in Hadoop 54

3.4 Summary 57

4. INCREMENTAL PROCESSING 58

4.1 A New Hash-based Platform 59

4.1.1 A Basic Hash Technique (MR-hash) 60
4.1.2 An Incremental Hash Technique (INC-hash) 62
4.1.3 A Dynamic Incremental Hash Technique (DINC-hash) 67

4.1.3.1 Analysis of the DINC Algorithm 70
4.1.3.2 Heuristic Improvement and the Marker Algorithm 74

4.1.4 Optimizing Other System Parameters 76

4.2 Prototype Implementation 77

4.3 Performance Evaluation 80

4.3.1 Small Key-state Space: MR-hash versus INC-hash 81
4.3.2 Large Key-state Space: INC-hash versus DINC-hash 83
4.3.3 Dynamic Hashing with Alternative Caching Algorithms 86
4.3.4 Dynamic Hashing under Coverage Requirements 90
4.3.5 Validation using Amazon EC2 91

4.4 Related Work 96
4.5 Summary 97

5. STREAM PROCESSING WITH LATENCY CONSTRAINTS 98

5.1 An Initial Benchmark Study 100
5.2 System Design 104

5.2.1 Overview of A Scalable Stream System 104
5.2.2 Extended MapReduce Architecture 108

5.3 Resource Planning 112

5.3.1 Model-driven Resource Planning 112
5.3.2 (μ, σ^2) of Per-Tuple Latency 115
5.3.3 (μ, σ^2) of Per-Window Latency 120
5.3.4 Quantiles of Latency 123
5.3.5 Job-Specific Statistics 124
5.3.6 Supporting Multiple MapReduce Rounds 125

5.4 Runtime Scheduling 126

5.4.1 Batch-level Scheduling 127
5.4.2 Tuple-level Cost-aware Scheduling 128
5.4.3 Optimization in Cost-aware Scheduling 136
5.4.4 Extension for Time Windows 137

5.5 Performance Evaluation 138

5.5.1 Latency-Aware Configuration 138
5.5.2 Latency-Aware Scheduling 143
5.5.3 Comparison to Other Systems 146
5.5.4 Summary 152

5.6 Related Work 152
5.7 Summary 154

6. CONCLUSIONS AND FUTURE WORK 156

6.1 Thesis Summary 156

6.1.1 Hadoop Benchmarking and Optimization 156
6.1.2 Incremental Processing 157
6.1.3 Stream Processing with Latency Constraints 158
6.1.4 Resulting System 159

6.2 Future Work 160

BIBLIOGRAPHY 161

LIST OF TABLES

Table **Page**

2.1 Design features and system parameters of existing data-parallel systems. 17

3.1 Workloads and their running time in the benchmark. 30

3.2 Average CPU cycles per node, measured by CPU seconds, in the map phase (256GB worldcup dataset). 35

3.3 Symbols used in Hadoop analysis. 41

4.1 Comparing optimized Hadoop (using sort-merge), MR-hash, and INC-hash. 85

4.2 Comparing sessionization to INC-hash with 0.5KB state, INC-hash with 2KB state, and DINC-hash with 2KB state. 85

4.3 Comparing INC-hash, DINC-Frequent and DINC-Marker in terms of reduce spill (GB). 87

4.4 Comparing INC-hash, DINC-Frequent, DINC-SS and DINC-Marker with the trigram counting workload over the Twitter dataset in terms of reduce spill (GB). 89

4.5 Comparing optimized Hadoop (using sort-merge), MR-hash, and INC-hash on EC2. 94

4.6 Comparing sessionization to INC-hash with 0.5KB state, INC-hash with 2KB state, and DINC-hash with 2KB state on EC2. 94

4.7 Comparing INC-hash, DINC-Frequent and DINC-Marker with the trigram counting workload over the Twitter dataset on EC2 in terms of reduce spill (GB). 95

5.1 Streaming workloads in Twitter feed and click analysis. 101

5.2 System parameters in modeling. 112

5.3 Latency breakdown of a map output tuple e (e' is the map input tuple that generates e). 116

5.4 Hardware specification in modeling. 117

5.5 Job-specific statistics required by the model. 124

5.6 Description and time complexity of basic operations over tree $\mathcal{T}$. 132

LIST OF FIGURES

Figure **Page**

2.1 Dependencies between system design features and system parameters for low-latency analytics. 14

3.1 Architecture of the Hadoop implementation of MapReduce. 29

3.2 Experimental results using the sessionization workload. 33

3.3 Task timeline using the inverted index construction workload. 36

3.4 Results for MapReduce Online using the sessionization workload. 39

3.5 Analysis of the tree of files created in multi-pass merge. 44

3.6 Validating our model against actual measurements in a Hadoop cluster. 49

3.7 Performance of optimized Hadoop based on our model. 52

3.8 Performance of MapReduce Online with pipelining of data. 54

4.1 MR-hash: hashing in mappers and two phase hash processing in reducers. 60

4.2 (Dynamic) Incremental Hash: monitoring keys and updating states. 62

4.3 Determining the number of buckets h given memory size B and key-state space size U. 65

4.4 Architecture of our Hadoop-based platform for incremental processing. 78

4.5 Progress report using hash implementations. 82

4.6 Progress report using hash implementations. 84

4.7 Disk spills of reducers in Twitter trigram counting when we tune the coverage requirement for the trigrams that occur more than 1000 times, under different input data sizes and reducer memory sizes. 90

4.8 Progress report using hash implementations on EC2. 92

4.9 Sessionization with DINC-hash on EC2. 93

5.1 A latency benchmark study of stock Hadoop and Incremental Hadoop. 103

5.2 A query modeled as a dataflow DAG. 105

5.3 MapReduce architecture extended with distributor & queues of mini batches. 109

5.4 An example of window latencies. 121

5.5 An example of tree $\mathcal{T}$. 133

5.6 Accuracy of the model for average latency. 139

5.7 Accuracy of the model for quantiles of latency. 140

5.8 Validity of the model-driven configuration optimization under average latency. 141

5.9 Validity of the model-driven configuration optimization under quantiles of latency. 142

5.10 Results of our model-driven configuration optimization. 143

5.11 Results of runtime scheduling. 144

5.12 Results of runtime scheduling. 145

5.13 Evaluation of Storm on latency and throughput. 147

5.14 Evaluation of Spark Streaming on latency and throughput. 149

5.15 Comparing Hadoop, INC-hash, Spark Streaming, Storm and our system on latency and throughput. 151

CHAPTER 1
INTRODUCTION

Today, data is being generated at an unprecedented rate – so much that 90% of the data in the world was generated in the last two years [44]. It is estimated that 2.5 quintillion (10^{18}) bytes of data are generated per day in the world from various sources including human-initiated data sources such as public web contents and social media, machine logs that record system and user activities, and sensors that monitor environment and human body conditions, etc. [88] The ability to process "big data" has become crucial to the information needs of many enterprise businesses, scientific applications, and governments. Recently, there have been increasing needs of processing data that is not only "big" but also "fast". Here "fast data" refers to high-speed real-time and near real-time data streams, such as Twitter feeds, search query streams, click streams, impressions, and system logs. For instance, a breaking news reporting service that monitors the Twitter firehose requires tweet feeds to be analyzed within seconds to detect hot topics and breaking news events [55]. As another example, Google's Zeitgeist pipeline ingests a continuous input of search queries and detects anomalous queries, which are spiking or dipping, again within seconds [4].

To handle both historical data and real-time data, many companies have to maintain multiple systems. However, recent real-world case studies point to the need for a unified data processing framework that supports big and fast data analytics, including analytical queries with different latency requirements. In particular, a recent study by Twitter's real-time search assistance team [65] shows the need to compute statistics across different temporal granularities with different latency tolerances e.g., evidence at the per-minute level to track fast moving, breaking events, as well as evidence accumulated across hours, days, or weeks for slower moving and tail queries. Using separate systems for batch processing and real-time processing causes not only code duplication, but also intensive manual work to partition the analytics workloads and determine which data is processed by which system.

Motivated by these observations, this thesis takes a further step towards a *general*, *unified* data processing framework that supports big and fast data analytics with different latency requirements. This thesis answers a fundamental question: “What are the necessary design features of a general, unified system for big and fast data analytics?” In order to build such a system, I propose to build on existing solutions on *data parallelism* and extend them with two new features: (1) *incremental processing* and (2) *stream processing with latency constraints*. In the following, I explain these concepts in more detail.

Data Parallelism is a fundamental mechanism for data processing at scale. It partitions a large dataset into smaller subsets, following either the storage order (physical partitioning) or a user-specified logical condition (logical partitioning), and then executes an analytic task in parallel over these subsets. MapReduce [25, 34, 47, 8, 68, 49, 54, 41] is a popular parallel processing model that embraces data parallelism, and has shown its scalability on thousands of machines. It also provides an easy-to-use programming model, and fault tolerance transparent to users. MapReduce and its most popular open-source implementation, Hadoop [94], have become the core of a large ecosystem of products for big data analytics. However, the MapReduce model is not well suited for time-sensitive workloads with latency requirements since it is primarily designed for batch processing. More concretely, MapReduce implementations require the entire dataset to be loaded into the cluster before running analytical queries and the query answers are only returned at the end of a long-running job, thereby incurring long latencies and making them unsuitable for processing “fast data”. Since MapReduce provides high scalability based on data parallelism and has gained tremendous popularity, in this thesis, I study how to support the following design features in the context of MapReduce. But my key ideas and techniques, once adapted, can be valuable to other big data systems based on data parallelism as well.

Incremental Processing means that computation is performed as soon as the relevant data becomes available. Incremental processing enables tuples to move quickly through a pipeline of operators, and thus is essential to returning query answers with low latency. Due to the need of quick movement of tuples between operators, incremental processing requires that no blocking operator should exist. However, blocking operators such as sort-merge are widely used in batch-processing systems, including the existing MapReduce systems. To respond to the needs of low-latency analytics, a variety of new systems have been developed, including Google's MillWheel [4] and Percolator [77], Twitter's Storm [92], Facebook's Ptail and Puma [13], Microsoft's Naiad [70] and Sonora [95], WalmartLabs' Muppet [55], IBM's System S [101], Yahoo's S4 [72], and academic prototypes such as Spark Streaming [99], StreamMapReduce [14], StreamCloud [35], SEEP [16] and TimeStream [79]. However, some systems [72, 92, 16, 70] do not provide built-in support of incremental processing, and thus impose the significant overhead of implementing incremental processing to users. Other systems mentioned above [14, 55, 95, 4, 99] provide mechanisms targeting at incremental processing, but with focus on the case when the memory is sufficiently large. Some of the systems can handle the situation of insufficient memory but only with simple approaches such as dropping and recomputing part of data [99], and simply spilling some data to a storage system backed by disks [55, 95, 4]. They do not provide any mechanism to maximize the portion of data processed using fast in-memory computation, and hence to perform incremental computation more efficiently.

In this thesis, I propose to fundamentally transform the batch processing paradigm in MapReduce systems into incremental processing, i.e., to move processing forward with newly arriving data items, and to do so efficiently even under constrained memory. Towards this goal, I propose a new purely hash-based framework with new hash algorithms and dynamic frequency analysis of data to support high-performance incremental processing under data parallelism.

Stream Processing with Latency Constraints is crucial to processing "fast data" in many applications [55, 65, 4]. For example, the latency constraints can be "news events should be returned within 3 seconds on average", and "99% anomalous queries should be identified within 500 milliseconds". However, the existing fast data systems [4, 14, 16, 35, 55, 70, 72, 79, 92, 95, 99] are "best-effort only", where system performance is determined by static parameter values set manually by system administrators. In practice, enterprise businesses cannot afford the manual work to find the optimal configuration that meets the latency constraint for each job.

In this thesis, I propose various techniques to support stream processing with latency constraints in a platform with data parallelism and incremental processing. I propose a number of necessary architecture changes to support stream processing, and augment the platform with new latency-aware model-driven resource planning and latency-aware runtime scheduling techniques to meet user-specified latency constraints while maximizing throughput.

In order to build a system with the above design features, I start with the open-source MapReduce implementation, Hadoop, due to its proven scalability based on data parallelism, as well as its many compatible software tools, and its broad user community. The work presented in this thesis answers the following questions: (1) Is Hadoop able to support incremental processing? (2) What are the necessary architecture changes in order to support incremental processing? (3) What are the additional design features needed to support stream processing with latency constraints? To address these questions, my thesis work is naturally divided into the following three pieces: Hadoop benchmarking and optimization, incremental processing, and stream processing with latency constraints. The contributions of each piece of work are discussed in detail in the subsequent section.

1.1 Thesis Contributions

1.1.1 Hadoop Benchmarking and Optimization

In the first piece of work, I aim to understand whether Hadoop, after tuning its parameters for optimization, is able to support incremental processing and why if not. In order to support incremental processing, where computation is performed as soon as the relevant data becomes available, a MapReduce system should avoid any blocking operations and also computational and I/O bottlenecks that prevent data from "smoothly" flowing through map and reduce phases on the processing pipeline.

With the questions and this guideline in mind, I conduct a thorough benchmarking study, evaluating existing MapReduce platforms including Hadoop and MapReduce Online (which performs pipelining of intermediate data [22]). The results reveal that the main mechanism for parallel processing used in these systems, based on a sort-merge technique, is subject to a significant I/O bottleneck as well as blocking: In particular, I find that the merge step is potentially blocking and can incur significant I/O costs due to intermediate data. Furthermore, MapReduce Online's pipelining functionality only redistributes workloads between the map and reduce tasks, and is not effective for reducing blocking or I/O overhead.

Building on these benchmarking results, I perform an in-depth analysis of Hadoop, using a theoretically sound analytical model to explain the empirical results. Given the complexity of the Hadoop software and its myriad of configuration parameters, I seek to understand whether the above performance limitations are inherent to Hadoop or whether tuning of key system parameters can overcome those drawbacks from the standpoint of incremental processing. The key results are two-fold: (1) It is shown that my analytical model can be used to choose appropriate values of Hadoop parameters, thereby reducing I/O and startup costs. (2) Despite a range of optimizations, the I/O bottleneck as well as blocking persist, and the reduce progress falls significantly behind the map progress, hence violating the requirements of efficient incremental processing. Both theoretical and empirical analyses show that the sort-merge implementation, used to support data parallelism, poses a fundamental barrier to incremental processing.

1.1.2 Incremental Processing

Based on the insight that the sort-merge implementation in the original MapReduce model poses a fundamental barrier to incremental processing, my next goal is to propose a new data analysis platform, based on MapReduce, that is geared for incremental processing. More concretely, I made two key architecture changes to Hadoop:

My first mechanism replaces the sort-merge implementation in Hadoop with a purely hash-based framework, which is designed to address the I/O bottleneck as well as the blocking behavior of sort-merge. I devise two hash techniques to suit different reduce functions, depending on whether the reduce function permits incremental processing or not. Besides eliminating the sorting cost from the map tasks, these hash techniques can provide fast in-memory processing of the reduce function when the memory reaches a sufficient size as determined by the workload and algorithm.

My second mechanism further brings the benefits of fast in-memory processing to workloads that require a large key-state space that far exceeds available memory. I propose both deterministic and randomized techniques to dynamically recognize popular keys and then update their states using a full in-memory processing path, both saving I/Os and enabling early answers for these keys. Less popular keys trigger I/Os to stage data to disk but have limited impact on the overall efficiency.

I implement all these techniques in Hadoop, which results in a prototype, namely Incremental Hadoop. Experiments on a range of workloads in click stream analysis and web document analysis show the following main results: (1) My hash techniques significantly improve the progress of the map tasks, due to the elimination of sorting, and given sufficient memory, enable fast in-memory processing of the reduce function. (2) For challenging workloads that require a large key-state space, my dynamic hashing mechanism significantly reduces I/Os and enables the reduce progress to keep up with the map progress, thereby realizing incremental processing. For instance, for sessionization over a click stream, the reducers output user sessions as data is read and finish as soon as all mappers finish reading the data in 34.5 minutes, triggering only 0.1GB internal data spill to disk in the job. In contrast, the original Hadoop system returns all the results towards the end of the 81 minute job, writing 370GB internal data spill to disk. (3) Further trade-offs exist between my hash-based techniques under different workload types, data localities, and memory sizes, with dynamic hashing working the best under constrained memory and most workloads.

1.1.3 Stream Processing with Latency Constraints

The revised MapReduce platform offers data parallelism and incremental processing. But are they enough for streaming analytical queries with stringent latency requirements? And if not, which additional design features are needed? These two questions will be addressed in my last piece of work.

My starting point is a thorough understanding of the sources of latency in a system supporting data parallelism and incremental processing. I conduct a benchmark study in the revised MapReduce platform, and it reveals that while incremental processing allows arriving tuples to be processed one-at-a-time, it does not guarantee the actual latency of processing each tuple in a large distributed system. (1) A key observation is that to enable streaming analytics with bounded latency of processing (e.g., 1 second) through a distributed system, it is crucial to determine the *degree of parallelism* (e.g., the number of processes per node) and *granularity of scheduling* (e.g., batching data items every 5ms for shuffling). Otherwise, upstream and downstream operators may process data at different speeds, causing substantial data accumulation in between. This reason, as well as using a large batch size as granularity for scheduling, will cause long wait time before tuples are processed or shuffled. The appropriate choices of those parameters vary widely among analytic tasks due to different computation needs – using the fixed values tuned for one workload is far from ideal for other workloads, often resulting in high latency of tuples moving through the system. (2) When the memory of a cluster is not large enough to process all data in memory, the tuples spilled to disk experience high latency because their processing is often deferred to a later phase (e.g., at the end) of the job.

These two key observations call for *job-specific resource planning* to select the appropriate parameter settings and *latency-aware scheduling* to determine which tuples to process and in what order to process them in order to keep latency low, both not being well-addressed by existing work. Tremendous engineering efforts have been invested in industry on the resource allocation for critical workloads. The recent development of Hadoop Yarn [93] separates resource management from data processing frameworks such as MapReduce, demonstrating the importance of resource management itself. However, Yarn only provides a friendly interface for users to set key system parameters, but cannot do so automatically for a given job. Recent fast data systems [4, 14, 16, 35, 55, 70, 72, 79, 92, 95, 99] are "best-effort only", where resource configuration does not take latency constraints or job characteristics as input. Instead, the configuration merely includes static parameter values set manually by system administrators. In practice, enterprise businesses cannot afford the manual work to find the optimal configuration for each job.

Different with existing work and effort, I solve the first issue by proposing a model-driven resource planning with two new unique features and the second issue by devising two runtime scheduling algorithms, as elaborated below.

Model-driven Resource Planning. To solve the first issue, I propose a model-driven approach to automatically determining the resource allocation plan for each job. The first unique aspect of my approach is that I consider performance, including both latency and throughput, in a holistic manner. A naive approach to minimizing latency may overprovision resources, e.g., giving all resources to push one tuple at a time through the distributed system, which limits throughput severely. Instead, I formulate the per-job resource planning problem as a constrained optimization problem: given a user analytic job and latency constraint $\mathcal{L}$, find a resource allocation plan to maximize throughput while subjecting the latency of results to $\mathcal{L}$. The second feature is the support of a variety of latency models, including per-tuple latency, per-window latency, and any quantiles associated with these latency distributions. Then any of these latency metrics can be subjected to the user latency constraint. To develop this collection of models, I identify and analyze the dominant components of latency in a complex distributed system. This involves two challenges: First, latency here covers a wide range of data processing behaviors, e.g., for processing individual tuples, producing windowed results from a set of tuples, and capturing the variation of each type of latency. Second, a distributed system based on data parallelism also exhibits complex system behaviors. My major contribution here lies in a clean extraction of dominant data processing and system-level behaviors from the Incremental Hadoop I previously implemented with a maze of complexity.

Latency-aware Scheduling. To solve the second issue, I further propose latency-aware scheduling at runtime to determine the set of tuples to process and the order to process them in order to maximize the number of results that meet the latency requirement, i.e., the *total utility*. Such runtime scheduling is helpful because at runtime, the workload characteristics may differ from those provided earlier to my model-driven resource planning, e.g., due to bursty inputs and change of computation costs under constrained memory. Thus, runtime selection and prioritization of tuples greatly affects the overall utility. While the problem of maximizing total utility is close to online scheduling problems in real-time operating systems [52, 21], these techniques either incur high time and space complexity for scheduling a large number of tuples, or do not consider tuple processing costs effectively. I propose two runtime scheduling algorithms, at batch-level and tuple-level, respectively, which consider both costs and deadlines of data processing. In particular, my tuple-level scheduling algorithm has provable results on the quality of runtime schedules and efficiency of the scheduling algorithm.

Evaluation using real-world workloads such as click stream analysis and tweet analysis show the following results: (1) My models can capture the trend of actual latency changes when tuning system parameters, with error rates within 15% for the average latency metric, and within 20% for 0.9 and 0.99-quantiles of latency. (2) My model-driven approach to resource planning can reduce the average latency from 10's of seconds in Incremental Hadoop to sub-second, with 2x-5x increase in throughput. (3) For runtime scheduling, my latency-aware tuple scheduling algorithm outperforms D^{over} [52], a state-of-the-art scheduling algorithm with provable optimality in the worse case, and can dramatically improve the number of tuples meeting the latency constraint, especially under constrained memory.

I finally compare our system to **Twitter Storm** [92] and **Spark Streaming** [99], two state-of-the-art distributed stream systems. For all workloads tested, my system offers *1-2 orders of magnitude* improvements over Storm and Sparking Streaming when considering both latency and throughput. As a proof-of-concept, my system, which is implemented in the general Hadoop framework, significantly outperforms custom distributed stream systems due to the key design features introduced in this work: our performance gains come from (1) the support of min-batches as the granularity for incremental processing and shuffling, while Spark Streaming lacks stable support of small batches and incremental processing of windows, and Storm incurs high scheduling overhead; (2) the job-specific resource allocation plans, which eliminate data accumulation and long wait in queues, which the other two systems cannot support.

1.1.4 System Development and Code Release

I have built a prototype of the platform for big and fast data analytics on Hadoop 0.20.1. And the resulting Hadoop-based prototype can run (1) in the original **batch mode**, fully supporting existing analytic tasks in enterprise businesses, or (2) in the new **streaming mode** where user-specified latency constraints are handled automatically by the system and used to guide resource planning and scheduling, with tremendous performance benefits over custom distributed stream systems. The code has been released as SCALLA 0.1 [62]. I also anticipate that while my implementation is in Hadoop, my key ideas and techniques, once adapted, can be valuable to other big data systems based on data parallelism.

1.2 Thesis Organization

The related work is discussed in Chapter 2. I perform benchmark study and system configuration optimization on existing MapReduce platforms in Chapter 3, and show that even the optimized MapReduce platform is still not well-suited for low-latency analytics due to the existence of a blocking operator and high I/O cost. In Chapter 4, I present a new platform for incremental processing that removes the blocking operator and minimizes I/O costs. To support stream processing with latency constraints, in Chapter 5, I further make necessary architectural changes and propose model-based configuration optimization as well as runtime scheduling techniques. I finally summarize the thesis and discuss the future work in Chapter 6.

CHAPTER 2
RELATED WORK

This chapter presents a survey of literature relevant to big data and fast data processing.

2.1 A Survey of Data-Parallel Systems

Data parallelism is a fundamental mechanism for data processing at scale. It partitions a large dataset into smaller subsets, following either the storage order (physical partitioning) or a user-specified logical condition (logical partitioning), and then executes an analytic task in parallel over these subsets. This section provides a survey of existing systems that support data parallelism, and compare them under the important design features for low-latency analytics. Before the discussion of the systems, we first introduce the key design features.

2.1.1 System Design Features for Low-Latency Analytics

We define a few system design features regarding low-latency analytics as follows.

- **Batch Processing** is the computation paradigm used to process a large collection of stored historical data. In batch processing, the results are usually returned at the end of a long-running job, and hence blocking operators that require all input data before emitting any output, such as sort-merge, are widely used.
- **Incremental Processing** means computation is performed as soon as the relevant data becomes available. With incremental processing, tuples can move quickly through a pipeline of operators. Incremental processing is a computation paradigm opposed to batch process in the sense that blocking operators should not exist. We use the following standard to determine that a system supports incremental processing: *The*

time between when an input tuple enters the system and when the tuple is consumed by the computation in any operator does not depend on the input size.

- **(Near) Real-time Latency Constraints** are supported if a system takes user-defined (near) real-time latency requirements into consideration and provides built-in mechanisms to satisfy the requirements.

Besides the above design features, several types of system parameters are closely related to low-latency analytics, and can be configured differently depending on the design features. The parameters are described below.

- **Granularity of Computation** is the data unit at which an operator schedules computation. There are several common granularities. (1) *Per-tuple*: an operator performs the computation of a tuple as soon as the tuple arrives. (2) *Mini-batch*: an operator collects input data, packs input data into small batches (e.g. every 5ms), and performs the computation for a batch at a time. (3) *Input Chunk*: an operator reads input data from a file system, such as HDFS, where input data is partitioned into physical chunks, and the operator performs the computation for a chunk at a time. (4) *All Data*: an operator waits until all input data is available, and then starts computation.
- **Granularity of Shuffling** is the data unit at which network transmission is scheduled. Common granularities are *Per-tuple*, *Mini-batch* and *Input Chunk*, which are defined similarly as in granularity of computation.
- **Degree of Parallelism** refers to the number of machines and the number of concurrent processes or threads per machine used to perform the computation of an operator.

Figure 2.1 shows the dependencies between the above system design features. (1) The support of (near) real-time latency constraints requires small granularities of computation and shuffling. The reason is that, in order to achieve low latency, tuples need to move quickly between operators, which can only happen when the granularities of computation

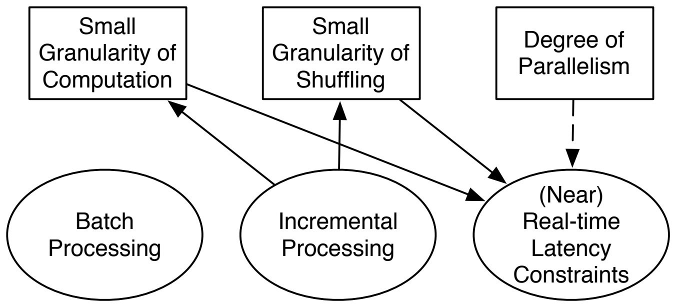

Small Granularity of Computation
Small Granularity of Shuffling
Degree of Parallelism
Batch Processing
Incremental Processing
(Near) Real-time Latency Constraints

Figure 2.1. Dependencies between system design features and system parameters for low-latency analytics.

and shuffling are small. (2) The support of (near) real-time latency constraints requires not only appropriate configuration of the granularities of computation and shuffling, but also appropriate configuration of the degree of parallelism. This is because the degree of parallelism affects the resource allocation among operators. As we will show in Chapter 5, inappropriate resource allocation may cause data accumulation between operators, and thus result in unbounded latency. (3) Small granularities of computation and shuffling require incremental processing. The reason is that the delay of computation at an operator may prevent the use of small granularities of computation and shuffling. For example, for a blocking operator, since it requires all input data before it emits any output, both the granularity of computation and the granularity of shuffling for the output can only be *All Data*. The dependency graph reveals the relationship between two system design features: *the support of latency constraints depends on incremental processing*. In this thesis, we first solve the problem of incremental processing, and then further study how to support latency constraints.

2.1.2 Comparison of Data-Parallel Systems

We next compare the existing data-parallel systems under the design features and key system parameters. Although the traditional parallel databases [28, 27] adopt data parallelism, we exclude them from the comparison due to their limited scalability. We start by a brief introduction of the state-of-the-art systems.

MapReduce [25] is a widely-used parallel processing model for big data analytics. At the API level, MapReduce simply includes two functions: The `map` function transforms input data into $\langle$key, value$\rangle$ pairs, and the `reduce` function is applied to each list of values that correspond to the same key. This programming model abstracts away complex distributed systems issues, thereby providing users with rapid utilization of computing resources. To achieve parallelism, at the system level, MapReduce essentially implements *"group data by key, then apply the reduce function to each group."* This computation model permits data parallelism because both the extraction of $\langle$key, value$\rangle$ pairs and the application of the `reduce` function to each group can be performed in parallel on many nodes. Besides this computation model, a MapReduce system also implements other functionalities such as scheduling, load balancing, and fault tolerance. An analytical query can be compiled (e.g., using a MapReduce-based query compiler [73]) into an acyclic dataflow graph, where each vertex is a MapReduce job consisting one `map` function and one `reduce` function, and an edge connects two jobs if one job takes the other job's output as input. **Hadoop [37]** is the most popular open-source implementation of MapReduce. The Hadoop Distributed File System (HDFS) handles the reading of job input data and writing of job output data. The unit of data storage in HDFS is a 64MB block by default (128MB in newer Hadoop releases) and can be set to other values during configuration. These blocks serve as the granularity for `map` computation and also control the granularity of data shuffling between mappers and reducers. Hadoop uses a sort-merge[1] technique to implement the group-by functionality [94] for parallel processing (Google's MapReduce system is reported to use a similar implementation [25], but further details are lacking due to the use of proprietary code). A large Hadoop-centered ecosystem for parallel processing, including Hive [43], Pig [78] and HBase [38], etc., has been built and applied widely in industry.

[1]Detailed description and analysis are available in Chapter 3.

MapReduce Online [22] improves MapReduce with pipelining of data. There are two district features for data pipelining from mappers to reducers. First, as each mapper produces output, it can push data eagerly to the reducers. The granularity of such data shuffling is controlled by a parameter. Second, an adaptive control mechanism is in place to balance work between mappers and reducers. For instance, if the reducers become overloaded, the mappers will write the output to local disks and wait until reducers are able to keep up again. MapReduce Online also allows reducers to periodically output snapshots (e.g., when reducers have received 25%, 50%, 75%, ..., of the data). This is done by performing the merge operation over the existing data for every snapshot. Hadoop Online Prototype (HOP) [36] is an open-source implementation of MapReduce Online based on Hadoop.

Spark [98] is an open-source, fault-tolerant parallel processing system tailored for in-memory computation. In Spark, data is stored in a distributed in-memory data abstraction, called Resilient Distributed Datasets (RDDs). Spark transforms an RDD to another RDD by user-defined coarse-grained transformations, including `map` and `reduce`, and other transformations such as `sample`, `distinct`, `union` and `intersection`. An analytical job consists of one or multiple transformations. Spark provides fault-tolerance by tracking the lineage of each RDD, and re-computing any lost partition of an RDD. **Spark Streaming [99]** is a functionality integrated in Spark that enables parallel stream processing. Spark Streaming periodically creates a mini-batch containing the streaming input data, and transforms stream processing into a sequence Spark batch jobs over input mini-batches.

Storm [92] is an open-source parallel stream processing system. A Storm cluster runs topologies that are similar to MapReduce jobs but can run continuously over streams. A topology is a dataflow graph of logical operators, which are called Spouts and Bolts. Spouts emit data as source streams of the system and Bolts consume input streams, do data processing, and possibly emit new streams to downstream operators. Storm supports shuffling between operators via various types of grouping, where field grouping is the same as the partitioning of map output in MapReduce. **MillWheel [4]** and **Naiad [70]** are two

Table 2.1. Design features and system parameters of existing data-parallel systems.

	Incremental Processing	**Granularity of Computation**	**Granularity of Shuffling**	**Latency Constraints**
Hadoop	No	Map: Input Chunk Reduce: All Data	Input Chunk	No
HOP	No	Map: Input Chunk Reduce: % of All Data	Mini-batch	No
Storm	Require User Implementition	Tuple	Tuple	No
Spark Streaming	Suboptimal	Mini-batch	Mini-batch	No
MillWheel	Suboptimal	Mini-batch	Mini-batch	No
Naiad	Suboptimal	Mini-batch	Mini-batch	No

parallel systems that also support continuous execution of a dataflow graph of operators. Compared to Storm, MillWheel provides a richer API and a stronger support of fault tolerance, while Naiad supports cyclic dataflow graphs in iterative workloads. Moreover, MillWheel and Naiad allow batching tuples for computation and shuffling, whereas Storm only supports per-tuple computation and shuffling.

We now compare the above data-parallel systems in terms of the design features and key system parameters as shown in Table 2.1. (1) *Incremental Processing.* As mentioned earlier, Hadoop and HOP adopt a blocking operator, sort-merge, to implement the partitioning in the MapReduce model. Hence, they do not support incremental processing. Storm provides a flexible API for parallel processing but imposes the complexity of the implementation of incremental processing to users. Spark Streaming, MillWheel and Naiad provide mechanisms for incremental processing, but focus on the case when the memory is sufficiently large. When the memory is insufficient, Spark Streaming drops part of the data that cannot fit in memory, and recompute the dropped data later, which incurs high cost due to repeated CPU computation, disk I/Os and network transmission. MillWheel simply stores data that cannot fit in memory in a data storage system, such as BigTable [19], without considering a more efficient way to perform in-memory incremental computation. Naiad lacks the detail about any mechanism dealing with insufficient memory. This thesis proposes efficient techniques for incremental processing with both sufficient and insufficient memory in Chapter 4. (2) *Granularity of Computation.* In Hadoop, due to the sort-merge operation, the execution of the `reduce` function cannot start before all data is received by the reducers. Thus, the reducers can only schedule computation at the granularity of all data. In HOP, since the output of snapshots is allowed, the reducers can schedule computation at the granularity of a pre-defined percentage of all data (e.g. every 25% of all data). As introduced earlier, in Storm, each operator processes a tuple at a time upon the arrival of a tuple. Spark Streaming, MillWheel and Naiad allow an operator to group tuples into mini-batches and process a batch at a time. (3) *Granularity of Shuffling.* The shuffling granularity in Hadoop is all the map output from an input chunk, while HOP allows more eager shuffling in smaller mini-batches. In Storm, the shuffling granularity is in individual tuples. In Spark Streaming, MillWheel and Naiad, the shuffling granularity is in mini-batches. (4) *(Near) Real-time Latency Constraints.* Hadoop and HOP do not support any latency constraint due to the lack of incremental processing. Although Spark Streaming, Storm, MillWheel and Naiad support incremental processing in some cases, they do not support user-defined latency constraints. In fact, all these existing systems are best-effort systems. They do not take latency constraints into consideration, whereas the system configuration merely includes static parameter values set manually by system administrators. We will further investigate the necessary system design features required by latency constraints besides incremental processing in Chapter 5.

Besides Spark Streaming, Storm, MillWheel and Naiad, a variety of other systems have been developed for parallel stream processing, such as Google's Percolator [77], Facebook's Ptail and Puma [13], Microsoft's Sonora [95], WalmatLabs' Muppet [55], IBM's System S [101], Yahoo's S4 [72], and academic prototypes such as StreamCloud [35], SEEP [16], TimeStream [79] and StreamMapReduce [14]. We now summarize the techniques used in these systems from different aspects in the subsequent sections.

2.2 Query Model and Query Language

A parallel stream system (PSS) is typically able to perform multiple continuous queries (jobs) in parallel. The logical model of a query varies from system to system, but usually can be generalized as a dataflow graph, where computation units (CUs) are interconnected by data streams [72, 92, 95, 4, 16, 79, 70]. A CU takes one or multiple input streams from data source or upstream CUs, and generates one or multiple output streams to downstream CUs or consumers outside the system. A CU can perform stateless computation such as filtering and transformation, or stateful computation such as window, group-by, aggregate and join.

PSSs are usually tailored for the shared-nothing architecture for high scalability. In order to handle high input rate, most systems provide both pipeline parallelism and data parallelism in the physical plan of execution. In pipeline parallelism, different CUs can run in parallel on different machines. In data parallelism, multiple instances (processes or threads) of a CU can run in parallel across different machines, with each instance performing a fraction of the workload of the CU. Both types of parallelism allow the execution of a query to scale out given more machines. Data parallelism requires that the input tuples are partitioned to multiple instances of a CU. Common partitioning schemes include random partitioning, key-based partitioning and broadcast. In random partitioning, a tuple is sent to a randomly selected CU instance. In key-based partitioning, the destination CU instance is determined by the value of a key attribute in the tuple. In broadcast, a tuple is sent to all instances of a CU. The partitioning scheme can be chosen based on the workload.

A system usually provides a low-level programming interface using a general programming language, such as Java or C++, to define the dataflow graph of a query and implement the functionalities of the CUs. The programming interface of a CU typically includes a set of user-defined callback functions. The most common callback is a function that is triggered by each tuple from the input stream. Other callbacks include functions triggered by timers [4] and functions triggered by system activities for fault tolerance [70, 16]. In these callbacks, a programmer can update the state of the CU instance if the CU is stateful, and call a system function to generate tuples to the output stream. In general, there are two types of low-level programming interfaces. In the first type, the system treats a CU instance as a black box, and user-defined functions can create and update an arbitrary form of state of the CU instance [72, 92, 16, 70]. This type of interface is flexible but imposes significant overhead to the programmer for functionalities such as fault tolerance, scaling out and memory management. In the second type, the system explicitly manages key-state pairs for a CU [14, 4]. The system partitions the key-state pairs to all the instances of the CU based on keys. The input tuples also have a key attribute. Each input tuple can only update the state that share the same key. The exposure of the computation state enables the system to provide more functionalities and optimizations that are transparent to the programmer. Some systems also support high-level query languages, such as LINQ [79]. A query written in a high-level language is usually automatically compiled into a job written in the low-level programming interface for execution.

2.3 Out-of-order Processing.

Since some operations are order-sensitive, such as time-based window, the system must handle out of order data due to delayed network delivery and different progress of workers in cluster computing. In particular, even if all the tuples arrive from external sources in time order or they are simply timestamped by the system in arrival order, after they are partitioned and assigned to different instances of a CU and processed on different nodes, there is unlikely any guarantee that the tuples will arrive at a downstream CU instance in time order. Since a CU running time-based window needs to identify the complete set of tuples that fall in a window, the system must have a way to tell the CU instance whether it has seen all the tuples in a time window.

There are two common solutions in the literature: One is to use punctuations [57] or low watermarks [4], which announce that at time t, an operator has seen all the data up to time $t - \delta$. If a CU instance receives such an announcement, it can process all the windows whose end times precede $t - \delta$. However, the most recent study on the MillWheel system [4] reports that out-of-order data can prevent low watermarks from advancing for large amounts of time.

The most advanced technique for out-of-order data [53] proposes to process significantly late data separately in order to reduce latency and buffering cost without dropping data. Consider an example of 30-second windows, and assume that time is marked using integers 1, 2, 3, ... Tuples that arrive before time 30 belong to the window $W_{30}^{v_1}$, where 30 is the end time of the window and v_1 is the version number. At time 30, the window closes and a result of $W_{30}^{v_1}$ is ready for output (or buffering). For the late tuples that arrive between time 30 and 60 but should belong to W_{30}, the system initiates a new window $W_{30}^{v_2}$ (which is possible if the operation permits incremental updates). At time 60, a result of $W_{30}^{v_2}$ is ready for output (or buffering), and so on.

2.4 Fault-tolerance.

Failure is common to see in a large group of machines. When a CU instance fails, a PSS should recover the state of the instance and guarantee correct output without causing high delay of output. Two types of approaches have been adopted by PSSs : (1) upstream backup and checkpointing [14, 95, 16, 4], and (2) lineage-based re-computation [79, 99]. In the first approach, each CU instance maintains a log of its output, and periodically creates a checkpoint of its state and the content in its output queues. A checkpoint is created asynchronously to a reliable storage such as a distributed file system. After that, the upstream CU instances are notified to prune the content older than the checkpoint in their output logs.

If the failure of a CU instance is detected, the PSS creates a new instance of the CU and restores the state and the content of the output queues from the most recent checkpoint of the failed instance. Then, the output logs of the upstream CU instances are replayed to the new instance to bring it up-to-date. This approach guarantees at-least-one delivery. The failed instance and the downstream instances may process some tuples more than once. The correctness of idempotent operators, such as min, max and union, is guaranteed. For non-idempotent operators, additional duplicate removal techniques are required [4]. In the second approach, the system keeps track of the lineage of the states during the processing, a dependency graph of the states and the operations that generate each state from its parent states. When the state of a CU instance is lost, it is recomputed from its parent states. If a parent state is also lost, the system recursively recover any lost states that are required based on the lineage graph. Based on either of the two approaches, in order to reduce the latency caused by recovery, techniques have been proposed to recover the state of a CU instance in parallel using multiple new instances [16, 99].

2.5 Elasticity

Due to the nature of continuous queries, a system should adapt to changes in the workload characteristics without interrupting the execution of the queries or causing high latency. Especially in the cloud environment, where users are charged in the pay-as-you-go manner, elasticity is highly desired to scale out to more machines when overload occurs and scale in to fewer machines when the machines are under utilized. For stateful operators [95, 16, 79], the basic idea is to abandon an instance of a CU that needs to scale out, and use fault-tolerance techniques to recover the abandoned instance with multiple new instances. [82] proposes techniques for stateless operators to scale out, with focus on finding the optimal number of processing threads for an operator.

We have introduced a variety of parallel stream processing systems from different technical aspects. In the subsequent sections, we discuss other relevant systems.

2.6 MapReduce

In this section, we briefly discuss the literature relevant to the MapReduce model.

Query Processing using MapReduce. [17, 47, 73, 76, 91, 97] has been a research topic of significant interest lately. To the best of our knowledge, none of these systems support incremental processing. Dryad [97] uses in-memory hashing to implement MapReduce group-by but falls back on the sort-merge implementation when the data size exceeds memory. Merge Reduce Merge [96] implements hash join using a technique similar to our baseline MR-hash, but lacks further implementation details. SCOPE [17] is a SQL-like declarative language designed for parallel processing with support of three key user-defined functions: process (similar to `map`), reduce (similar to `reduce`) and combine (similar to a join operator). SCOPE provides the same functionality as Merge Reduce Merge and does not propose new hash techniques beyond the state-of-the-art. Several other projects are in parallel to our work: The work in [8] focuses on optimizing Hadoop parameters and ParaTimer [68] aims to provide an indicator of remaining time of MapReduce jobs. Neither of them improves MapReduce for incremental computation.

Cost Models for MapReduce. Recent work presented models to predict total running time of a MapReduce job over stored data based on analysis of CPU, I/O and network costs or based on training over profiles of previous jobs [40, 41, 42, 45, 30]. These models differ from ours as we aim to capture latency, which relates to a different set of system parameters and concerns, such as the input rate, queue sizes, and latency in each step. ParaTimer [68] is a progress indictor for MapReduce jobs, which identifies map and reduce tasks on a query's critical path. It serves a different purpose from our goal of modeling latency. Zeitler et al. [100] proposed a model of total CPU cost for distributed continuous queries, a different goal from ours.

Computation Models for MapReduce. Karloff et al. [49] suggest a theoretical abstraction of MapReduce. The abstraction employs a model that is both general and sufficiently simple such that it encourages algorithm research to design and analyze more sophisticated MapReduce algorithms. However, this model is based on simple assumptions that are not suitable for system research like our study. First, no distinction is made between disk and main memory. Second, the model is only applicable to the case that each machine has memory $O(n^{1-\epsilon})$ where n is the size of entire input and ϵ is a small constant. Third, the model assumes that all map tasks have to complete before the beginning of the reduce phase, which makes the model invalid for incremental computation.

2.7 Stream Database Systems

Stream databases [1, 15, 18, 69] laid a foundation for stream processing including window semantics, optimized implementations, and out-of-order data processing. Our work leverages state-of-the-art techniques for windowed operations and out of order processing in the new context of the MapReduce cluster computing. Techniques for QoS including scheduling and load shedding have been studied in Aurora/Borealis [15, 7, 1, 59, 90, 89]. However, scheduling in Aurora [15] considers latency only based on CPU costs and selectivity of operators in the single-machine environment, whereas our resource planning considers many more factors in a distributed system and uses constrained optimization to support both latency and throughput. Load shedding in Borealis [89] allocates resources in a distributed environment by solving a linear optimization problem, but only maximizes throughput without considering latency and only supports pipeline parallelism without data parallelism.

2.8 Parallel Database Systems

Parallel databases [28, 27] require special hardware and lacked sufficient solutions to fault tolerance, hence having limited scalability. Their implementations use hashing intensively. In contrast, our work leverages the massive parallelism of MapReduce and extends it to incremental processing and low-latency analytics. We use MR-hash, a technique similar to hybrid hash used in parallel databases [28], as a baseline. Our more advanced hash techniques emphasize incremental processing and in-memory processing for hot keys in order to support low-latency analytics.

Emerging parallel databases [2, 3, 12, 32] use MapReduce for query processing, leveraging its scalability and fault tolerance, but do not support incremental processing or low-latency analytics.

2.9 Other Systems

DEDUCE [54] extends System S with the capability of executing MapReduce jobs, but for batch processing. CBP [61] supports stateful bulk processing for incremental analytics, but not low-latency streaming analytics (e.g., with running time of 10's to 100's of minutes). Photon [5] is designed particularly for joining data streams in real-time, but focuses on issues other than efficient incremental processing and supporting latency constraints.

CHAPTER 3
HADOOP BENCHMARKING AND OPTIMIZATION

In this chapter, we answer the following research question: “Are the existing MapReduce implementations able to support incremental processing?” Incremental processing means that computation is performed as soon as the relevant data becomes available. Incremental processing enables tuples to move quickly through a pipeline of operators, and thus is essential to returning query answers with low latency.

We first conduct a thorough benchmarking study, evaluating existing MapReduce platforms including Hadoop and MapReduce Online (which performs pipelining of intermediate data [22]). The results reveal that the main mechanism for parallel processing used in these systems, based on a sort-merge technique, is subject to a significant I/O bottleneck as well as blocking: In particular, we find that the merge step is potentially blocking and can incur significant I/O costs due to intermediate data. Furthermore, MapReduce Online’s pipelining functionality only redistributes workloads between the map and reduce tasks, and is not effective for reducing blocking or I/O overhead.

Next, building on these benchmarking results, we perform an in-depth analysis of Hadoop, using a theoretically sound analytical model to explain the empirical results. Given the complexity of the Hadoop software and its myriad of configuration parameters, we seek to understand whether the above performance limitations are inherent to Hadoop or whether tuning of key system parameters can overcome those drawbacks from the standpoint of incremental processing. The key results are two-fold: (1) It is shown that our analytical model can be used to choose appropriate values of Hadoop parameters, thereby reducing I/O and startup costs. (2) Despite a range of optimizations, the I/O bottleneck as well as blocking persist, and the reduce progress falls significantly behind the map progress, hence violating the requirements of efficient incremental processing. Both theoretical and empirical analyses show that the sort-merge implementation, used to support data parallelism, poses a fundamental barrier to incremental processing.

This chapter is organized as follows. We first provide a technical background of MapReduce in Section 3.1. We then describe our benchmark results of the MapReduce systems in Section 3.2. We show the optimization of Hadoop configuration and the analysis of the results in Section 3.3. We finally conclude in Section 3.4.

3.1 Background

To provide a technical context for the discussion in this chapter, we begin with background on MapReduce, followed by the overview of Hadoop, the most popular open-source implementation of MapReduce.

3.1.1 Overview of MapReduce

At the API level, the MapReduce *programming model* simply includes two functions: The `map` function transforms input data into $\langle$key, value$\rangle$ pairs, and the `reduce` function is applied to each list of values that correspond to the same key. This programming model abstracts away complex distributed systems issues, thereby providing users with rapid utilization of computing resources. As an example, consider how we would parse a click stream to find the most visited pages. More specifically, we want to count how many times each page has been visited. Suppose the schema for a visits table is (timestamp, user, url). Consider the following SQL query:

```
SELECT COUNT(*) FROM visits GROUP BY url;
```

Page clicks are grouped by url and then aggregated using `COUNT` for each url. Now consider the equivalent MapReduce job for computing page frequencies below. The map function emits a new $<$url, count$>$ tuple for each visit in the click stream, where the count is 1. The reduce function then groups these tuples by url and sums the number of visits to each url.

```
function map(rid, visit) {
   emit(visit.url, 1);
}

function reduce(url, iterator<int> count) {
   int total = 0;
   foreach count c
      total += c;
   emit(url, total);
}
```

To achieve parallelism, the MapReduce system essentially implements "*group data by key, then apply the reduce function to each group*". This *computation model*, referred to as MapReduce group-by, permits parallelism because both the extraction of $\langle$key, value$\rangle$ pairs and the application of the reduce function to each group can be performed in parallel on many nodes. The system code of MapReduce implements this computation model (and other functionality such as load balancing and fault tolerance).

The MapReduce program of an analytical query includes both the map and reduce functions compiled from the query (e.g., using a MapReduce-based query compiler [73, 43]) and the MapReduce system's code for parallelism.

3.1.2 Overview of the Hadoop Implementation

We consider Hadoop, the most popular open-source implementation of MapReduce, in our study. Hadoop uses block-level scheduling and a sort-merge technique [94] to implement the group-by functionality for parallel processing (Google's MapReduce system is reported to use a similar implementation [25], but further details are lacking due to the use of proprietary code).

The Hadoop Distributed File System (HDFS) handles fault tolerance and replication for reading job input data and writing job output data. By default, the unit of data storage in

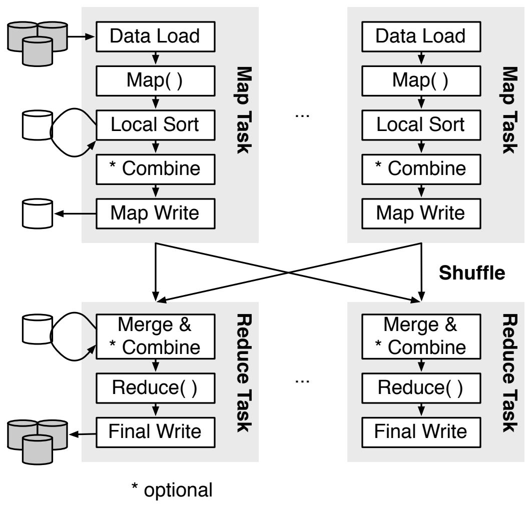


Figure 3.1. Architecture of the Hadoop implementation of MapReduce.

HDFS is a 64MB block and can be set to other values during configuration. These blocks serve as the task granularity for MapReduce jobs.

Given a query, its MapReduce job is assigned m map tasks (mappers) and r reduce tasks (reducers) concurrently on each node. As Figure 3.1 shows, each mapper reads a block of input data, applies the map function to extract key-value pairs, then assigns these data items to partitions that correspond to different reducers, and finally sorts the data items in each partition by the key. Hadoop currently performs a *block-level sort* on the compound (partition, key) to achieve both partitioning and sorting in each partition. Given the relatively small block size, a properly-tuned buffer will allow such sorting to complete in memory. Then the sorted map output is written to a file using synchronous I/O. A mapper completes after its output has been persisted for fault tolerance.

Map output is then shuffled to the reducers (in the *shuffling* phase). To do so, reducers periodically poll a centralized service asking about completed mappers and once notified, requests data directly from the completed mappers (*pull*-based communication). Under normal circumstances, this data transfer happens soon after a mapper completes and so this data is often available in the mapper's memory.

Table 3.1. Workloads and their running time in the benchmark.

	Click Streams			**Web Documents**
Setting	*Sessionization*	*Page frequency*	*Per-user count*	*Inverted index*
Input data	256GB	508GB	256GB	427GB
Map output data	269GB	1.8GB	2.6 GB	150GB
Reduce spill data	370GB	0.2GB	1.4 GB	150GB
Intermediate/input	250%	0.4%	1.0%	70%
Output data	256GB	0.02GB	0.6GB	103GB
Map tasks	3,773	7,580	3,773	6,803
Reduce tasks	40	40	40	40
Completion time	76 min.	40 min.	24 min.	118 min.

Over time, a reducer collects pieces of sorted output from many completed mappers. Unlike before, this data cannot be assumed to fit in memory for larger workloads. As the reducer's buffer fills up, these sorted pieces of data are merged and written to a file on disk. A background thread merges these on-disk files progressively whenever the number of such files exceeds a threshold F (in a so-called *multi-pass merge* phase). When a reducer has collected all of the map output, it will perform a multi-pass merge if the on-disk files exceed F; otherwise, it will perform a final merge to produce all key-value pairs in sorted order of the key. Over the sorted file, the reducer identifies each list of values sharing the same key and then applies the reduce function to the list. The output of the reduce function is written back to HDFS.

Finally, when the reduce function is commutative and associative, a combine function (typically sharing the code with the reduce function) is applied right after the map function, as shown in Figure 3.1, to perform partial aggregation and reduce the size of map output. It can be further applied in a reducer when its data buffer fills up.

3.2 Benchmarking and Analysis

We next introduce our benchmark study in detail.

3.2.1 Experimental Setup

We consider two applications that require incremental processing in benchmarking: click stream analysis which represents workloads for stream processing, and web document analysis which represents workloads for one-pass analysis over stored data. The workloads tested are summarized in Table 3.1.

In click stream analysis, an important task is sessionization, which reorders click logs into individual user sessions. Its MapReduce program employs the map function to extract the url and user id from each click log, then groups click logs by user id, and implements the sessionization algorithm in the reduce function. A key feature of this task is a large amount of intermediate data due to the reorganization of all click logs by user id. Another task in click stream analysis is page frequency counting. As a simple variant on the canonical word counting problem, it counts the number of visits to each url. A similar task counts the number of clicks that each user has made. For such counting problems, a combine function can be applied to significantly reduce the amount of intermediate data. For this application, we use the click logs from the World Cup 1998 website[1] and replicate it to larger sizes as needed.

The second application is web document analysis. A key task is inverted index construction, in which a large collection of web documents (or newly crawled web documents) is parsed and an inverted index on the occurrences of each word in those documents is created. In its MapReduce program, the map function extracts (word, (doc id, position)) pairs and the reduce function builds a list of document ids and positions for each word. The intermediate data is typically smaller than the document collection itself, but still of a substantial size. Other useful tasks in this application involve word frequency analysis, which are similar to page frequency analysis mentioned above, hence omitted in Table 3.1.

[1]http://ita.ee.lbl.gov/html/contrib/WorldCup.html

For this application, we use the 427GB GOV2 document collection created from an early 2004 crawl of government websites.[2]

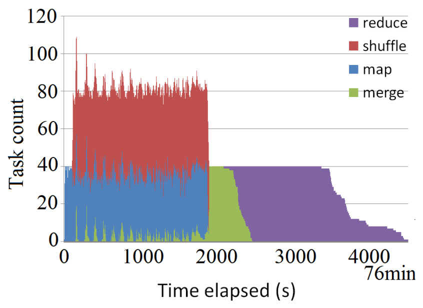
The Hadoop configuration mainly used the default settings with a few changes. We ran the NameNode and JobTracker daemons on the head node and ran DataNode and TaskTracker daemons on each of the 10 compute nodes. The HDFS block size was 64MB. HDFS replication was turned down to 1 from the default 3. The map output buffer was tuned for each workload to ensure there were no spills to disk. JVM reuse was enabled. The JVM heap size was set to 1GB.

A variety of tools are used for profiling, all of which have been packaged into a single program for simplicity. This program launches standard utilities such as `iostat` and `ps`, and logs the output to a file. We use the logged information to track metrics such as disk utilization and system CPU utilization. Hadoop-specific plots such as the task history were created by a publicly available parser.

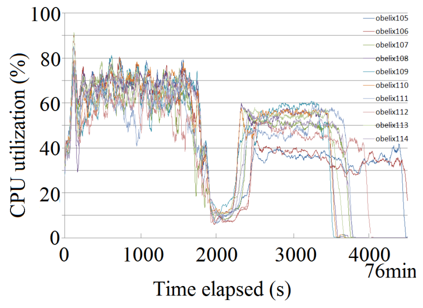
3.2.2 Result Analysis

Table 3.1 shows the running time of the workloads as well as the sizes of input, output, and intermediate data in our benchmark. Our analysis below focuses on the sessionization workload that involves the largest amount of intermediate data. We comment on the results of other workloads in the discussion whenever appropriate. Figure 3.2(a) shows the task timeline for the sessionization workload, i.e., the number of tasks for the four main operations in its MapReduce job: *map* (including sorting), *shuffle*, *merge* (the multi-pass part), and *reduce* (including the final scan to produce a single sorted run). As can be seen, time is roughly evenly split between the map and reduce phases, with a substantial merge phase in between. Also note that some periodic background merges take place even before all map tasks complete. When the intermediate data is reduced as in other workloads, first the merge phase shrinks and then the reduce phase also shrinks.

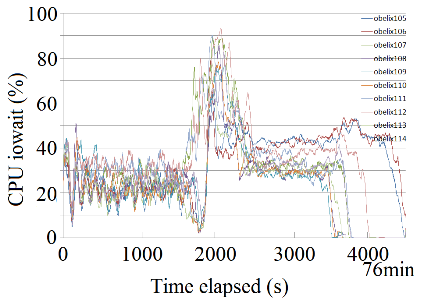
[2]http://ir.dcs.gla.ac.uk/test_collections/gov2-summary.htm



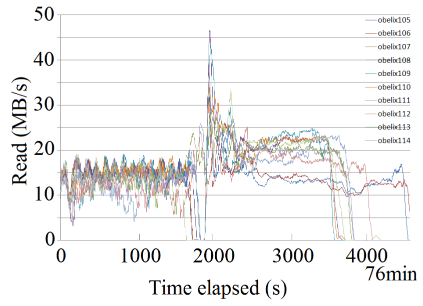
(a) Task timeline.



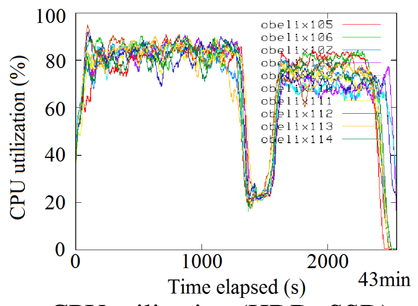
(b) CPU utilization.



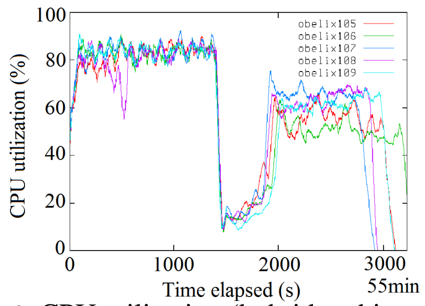
(c) CPU iowait.



(d) Bytes read.



(e) CPU utilization (HDD+SSD).



(f) CPU utilization (hybrid architecture).

Figure 3.2. Experimental results using the sessionization workload.

Cost of Parsing. A potential CPU bottleneck can be parsing line-oriented flat text files into the data types that map functions expect. To investigate this possibility, we prepared two different formats of the same data to use as input for the sessionization workload. The first format is the original line-oriented text files, leaving the task of extracting user ids to a regular expression in the map function. The second format is the same data preprocessed into Hadoop's `SequenceFile` binary format, allowing the map function to immediately operate on the data without having to do any parsing. We ran the sessionization workload on these two inputs and observed almost no difference in either running time or CPU utilization between the jobs. We therefore concluded that input parsing is a negligible overall cost.

Cost of Map Output. A potential I/O bottleneck can be the writes of map output to disk using synchronous I/O, required for fault tolerance in MapReduce. In our benchmark, we observed that although each map task did block while performing this write, it did not take up a large portion of a map task's lifetime. In the sessionization workload with a large amount of map output data, these writes took 1.3 seconds on average, while the average map task running time took 21.6 seconds. This 6% time did not make a significant contribution to a map task's running time relative to other parts. Furthermore, the recent MapReduce Online system [22] proposes to pipeline map output to the reducers and persists the data using asynchronous I/O. Hence, it can be used as a solution if the map output may be observed as an I/O bottleneck elsewhere.

Overhead of Sorting. Recall from Section 3.1 that when a map task finishes processing its input block, the key-value pairs must be partitioned according to different reducers and key-value pairs in each partition must be sorted to facilitate the merge in reducers. Hadoop accomplishes this task by performing a sort on the map output buffer on the compound of (partition, key).

First, we observe from Figure 3.2(b) that CPUs are busy in the map phase. It is important to note that the map function in the sessionization workload is relatively CPU light: it parses each click log into user id, timestamp, url, etc., and emits a key-value pair where the key is the user id and the value contains other attributes. The rest of cost in the map phase is attributed to sorting of the map output buffer. To quantify the costs of the map function and sorting, we performed detailed profiling of CPU cyles consumed by each, as shown in Table 3.2. In the sessionization workload, roughly 61% of CPU cycles were consumed by the map function while 39% was by sorting. In the per-user click counting workload, the map function simply emits pairs in the form of (user id, "1"), and up to 48% of CPU cycles were consumed by sorting these pairs. We further note that if we expedite click log

Table 3.2. Average CPU cycles per node, measured by CPU seconds, in the map phase (256GB worldcup dataset).

	Sessionization	*Per-user count*
Map function (%)	566 sec. (61%)	440 sec. (52%)
Sorting (%)	369 sec. (39%)	406 sec. (48%)

parsing in the map function using the recent proposal of mutable parsing [47], the overhead of sorting will be even more prominent in the map phase.

Conclusion: Sorting of map output can introduce a significant CPU overhead, due to the use of the sort-merge implementation of the group-by operation in MapReduce.

Overhead of Merging. As map tasks complete and their output files are shuffled to the reducers, each reducer writes these files to disk (since there is not enough memory to hold all of them) and performs multi-pass merge: as soon as the number of on-disk files reaches a configurable threshold, it merges these files to a larger file and writes it back to disk. Such a merge will be triggered next time when the reducer sees this number of files on disk. This process continues until all map tasks have completed and the reducer has brought the number of on-disk files down to the threshold. It completes by merging these on-disk files and feeding sorted data directly into the reduce function.

In the sessionization workload, the overhead of multi-pass merge is particularly noticeable when most map tasks have completed. In the CPU utilization plot in Figure 3.2(b), there is an extended period (from time 1800 to 2400) where the CPUs are mostly idle. While CPUs could be idle due to both disk I/O and network I/O, the CPU iowait graph in Figure 3.2(c) shows that it is largely due to outstanding disk I/O requests, and the graph in Figure 3.2(d) shows a large number of bytes read from disk in the same period. All of these observations match the merge activities shown between the map and reduce phases in the task timeline plot in Figure 3.2(a).

Overall, multi-pass merge is a blocking operation. The reduce function cannot be applied until this operation completes with all the data arranged into a single sorted run.

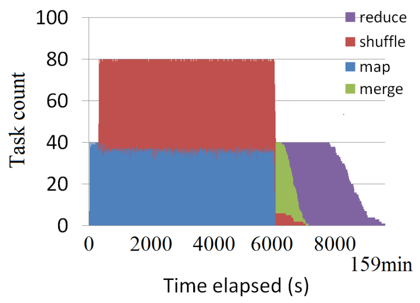


Figure 3.3. Task timeline using the inverted index construction workload.

This blocking effect causes low CPU utilization when most map tasks complete and prevents any answer from being returned by reducers for an extended period.

Moreover, the multi-pass merge operation is also I/O intensive. Our profiling tool shows that in sessionization, the reducers read and write 370GB data in the multi-pass merge operation while the input data has only 256 GB, as shown in Table 3.1. The inverted index workload incurs a somewhat reduced but still substantial I/O cost of 150GB data in this operation. As shown in Figure 3.3, the blocking merge phase is present in this workload as well. Progress is stopped until local intermediate data is merged on each node. In simpler workloads, such as counting the number of clicks per user, there is an effective combine function to reduce the intermediate data size. However, it is interesting to observe from Table 3.1 that even if there is ample memory to perform in-memory processing, the multi-pass merge still causes I/O, e.g., 1.4GB spill from the reducers. This is because when the memory fills up, each reducer applies the combine function to the data in memory but still writes the data to disk waiting for all future data to produce a single sorted run.

Conclusion: The multi-pass merge operation is blocking. It is I/O intensive for workloads with large amounts of intermediate data. It may still cause I/O even if there is enough memory to hold all intermediate data.

3.2.3 Architectural Improvements

We next explore some architectural choices to investigate whether these changes can eliminate the blocking effect and the I/O bottleneck observed in the previous benchmark. The motivation is that when given a substantial amount of intermediate data, the disk utilization stays high for most time of a MapReduce job (e.g., over 90% in the sessionization workload). This is because the disk on each node not only serves the input data from HDFS and writes the final output to HDFS, but also handles intermediate data including the map output and the reduce spill in the multi-pass merge. Given a mix of requests from different MapReduce operations, the disk is often maxed out and subject to random I/Os.

Separate Storage Devices. One architectural improvement is to employ multiple devices per node for storage, thereby reducing disk contention in MapReduce operations. In this experiment, in addition to the existing hard disk, we add a solid state drive (64GB Intel SSD) to each node in the cluster. We use the hard disk to handle the input and output with HDFS and use the smaller, but faster, SSD to hold all intermediate data. This way, reading input data from HDFS and managing the intermediate data can proceed in parallel. In addition, the writes of map output and the reads/writes for multi-pass merge can benefit from the fast random access offered by the SSD.

We show the CPU utilization (among many other measurements) of the sessionization workload in Figure 3.2(e). The main observations include the following. Extra storage devices help reduce the total running time, from 76 minutes to 43 minutes for sessionization. Detailed profiling shows that roughly 2/3 of the performance benefit comes from having an extra storage device, and about 1/3 of it comes from the SSD characteristics themselves. However, there is still a significant period where the CPU utilization is low, demonstrating that the multi-pass merge continues to be blocking and involving intensive I/Os.

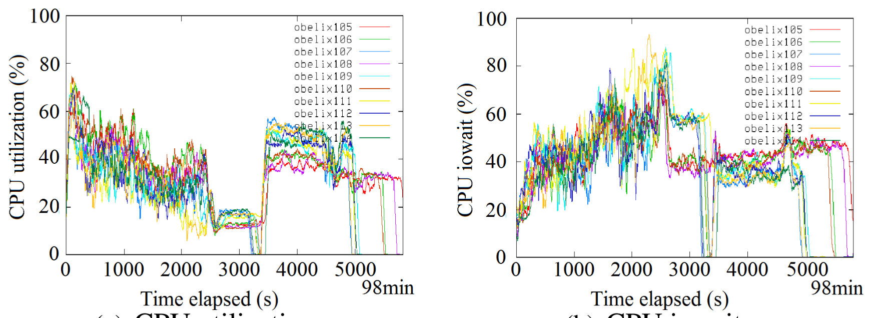
A Separate Distributed Storage System. An alternative way to address the disk contention problem is to use separate systems to host the distributed storage and MapReduce computation. This is analogous to Amazon's Elastic MapReduce where the S3 system handles distributed storage and the EC2 system handles MapReduce computation with its local disks reserved for the use of intermediate data. This comes at the price of data locality though; tasks will no longer be able to be scheduled on the same nodes where data resides and so this architecture will incur additional network overhead. In our experiment, we simulate two subsystems by allocating 5 nodes to host the distributed storage and 5 nodes to serve as compute nodes for MapReduce. We reduce the input data size accordingly to keep the running time comparable to before.

Similar to the previous experiment, the separation of the distributed storage system helps reduce the running time of sessionization from 76 minutes to 55 minutes (which, however, does not have the benefits of SSDs). More importantly, the CPU utilization plot in Figure 3.2(f) shows that the issues of blocking and intensive I/O remain, which agrees with the previous experiment.

Conclusion: Architectural improvements can help reduce contention in storage device usage and decrease overall running time. However, they do not eliminate the blocking effect or the I/O bottleneck observed about the sort-merge implementation of MapReduce.

3.2.4 MapReduce Online

We finally consider a recent system called MapReduce Online that implements a Hadoop Online Prototype (HOP) with pipelining of data [22]. This prototype has two distinct features: First, as each map task produces output, it can push data eagerly to the reducers. The granularity of such data transmission is controlled by a parameter. Second, an adaptive control mechanism is in place to balance work between mappers and reducers. For instance, if the reducers become overloaded, the mappers will write the output to local disks and wait until reducers are able to keep up again. A potential benefit of HOP is that with pipelining, reducers receive map output earlier and can begin multi-pass merge earlier, thereby reducing the time required for the merge work after all mappers finish.



(a) CPU utilization. (b) CPU iowait.

Figure 3.4. Results for MapReduce Online using the sessionization workload.

However, it is important to note that HOP adds pipelining to an overall blocking implementation of MapReduce based on sort-merge. As is known in the database literature, the sort-merge implementation of group by is an inherently blocking operation. HOP has a minor extension to periodically output snapshots (e.g., when reducers have received 25%, 50%, 75%, ..., of the data). This is done by repeating the merge operation for each snapshot. This is not real incremental computation desired in stream processing, and may incur a significant I/O overhead in doing so. Furthermore, such pipelining does not reduce CPU and I/O overhead but only redistributes workloads between mappers and reducers.

Figure 3.4 shows some initial results of MapReduce Online using the sessionization workload. The most important observation is that the CPU utilization plot shows a similar pattern of low values in the middle of the job. While CPU can be idle due to both I/O wait and network wait (given the different communication model used in MapReduce Online), the CPU iowait graph again shows a spike in the middle of the job. Hence, our previous observations of blocking and I/O activity due to multi-pass merge still hold here.

There are several subtle differences from the previous results of benchmarking Hadoop. The total running time is actually longer using MapReduce Online. A possible explanation for this difference is that MapReduce Online is based off an older version of Hadoop, 0.19.2, whereas we benchmarked using 0.20.0. Any performance optimizations made during this time will only be present in the newer version. Another possible reason is that MapReduce

Online transmits map output eagerly in finer granularity and hence increases network cost, which in turn causes lower CPU utilization. Another thing to note is that the CPU utilization in the map phase when running HOP is lower than when running on stock Hadoop. We verified that the total number of CPU cycles consumed in the map phase are similar across both implementations by observing that HOP spends a greater amount of time in the map phase, with a somewhat reduced level of CPU utilization. Finally, this prototype moves some of the sorting work to reducers, which may also affect the CPU utilization in different phases of the job. In our ongoing work, we will continue benchmarking MapReduce Online, including the use of other workloads, to better explain its behavior.

3.2.5 Summary of Results

In this section, we benchmarked Hadoop and MapReduce Online which both use the sort-merge implementation of the group by operation in MapReduce. Our goal was to answer the question that we raised at the beginning of the study: Do current MapReduce systems satisfy the requirements for incremental processing? Our benchmarking results can be summarized as follows.

- ▶ The sorting step of the sort-merge implementation incurs high CPU cost, hence unsuitable for fast in-memory processing.
- ▶ Multi-pass merge in sort-merge is blocking and can incur high I/O cost given substantial intermediate data, hence not suitable for incremental processing or fast in-memory processing.
- ▶ Using extra storage devices and alternative storage architectures do not eliminate blocking or the I/O bottleneck.
- ▶ The Hadoop Online Prototype with pipelining does not eliminate blocking, the CPU bottleneck, or the I/O bottleneck.

Table 3.3. Symbols used in Hadoop analysis.

Symbol	**Description**
(1) System Settings	
R	Number of reduce tasks per node
C	Map input chunk size
F	Merge factor that controls how often on-disk files are merged
(2) Workload Description	
D	Input data size
K_m	Ratio of output size to input size for the map function
K_r	Ratio of output size to input size for the reduce function
(3) Hardware Resources	
N	Number of nodes in the cluster
B_m	Output buffer size per map task
B_r	Shuffle buffer size per reduce task
(4) Symbols Used in the Analysis	
U	Bytes read and written per node, $U = U_1 + \ldots + U_5$ where U_i is the number of bytes of the following types 1: map input; 2: map internal spills; 3: map output; 4: reduce internal spills; 5: reduce output
S	Number of sequential I/O requests per node
T	Time measurement for startup and I/O cost
h	Height of the tree structure for multi-pass merge

3.3 Optimizing Hadoop

Building on our previous benchmarking results, we perform an in-depth analysis of Hadoop in this section. Our goal is to understand whether the performance issues identified by our benchmarking study are inherent to Hadoop or whether they can be overcome by appropriate tuning of key system parameters.

3.3.1 An Analytical Model for Hadoop

The Hadoop system has a large number of parameters. While our previous experiments used the default setting, we examine these parameters more carefully in this study. After a nearly year-long effort to experiment with Hadoop, we identified several parameters that impact performance from the standpoint of incremental processing, which are listed in Part (1) of Table 3.3. Our analysis below focuses on the effects of these parameters on I/O and startup costs. We do not aim to model the actual running time because it depends on numerous factors such as the actual server configuration, how map and reduces tasks are interleaved, how CPU and I/O operations are interleaved, and even how simultaneous I/O requests are served. Once we optimize these parameters based on our model, we will evaluate performance empirically using the actual running time and the progress with respect to incremental processing.

Our analysis makes several assumptions for simplicity: The MapReduce job under consideration does not use a combine function. Each reducer processes an equal number of ⟨key, value⟩ pairs. Finally, when a reducer pulls a mapper for data, the mapper has just finished so its output can be read directly from its local memory. The last assumption frees us from the onerous task of modeling the caching behavior at each node in a highly complex system.

3.3.1.1 Modeling I/O Cost in Bytes

We analyze the I/O cost of the *existing* sort-merge implementation of Hadoop. We first consider the I/O cost in terms of the number of bytes read and written. Our main result is summarized in the following proposition.

Proposition 3.3.1. *Given the workload description (D, K_m, K_r) and the hardware description (N, B_m, B_r), as defined in Table 3.3, the I/O cost in terms of bytes read and written in a Hadoop job is:*

$$\begin{aligned} U =& \frac{D}{N} \cdot (1 + K_m + K_m K_r) + \frac{2D}{CN} \cdot \lambda_F(\frac{CK_m}{B_m}, B_m) \cdot \mathbb{1}_{[C \cdot K_m > B_m]} \\ &+ 2R \cdot \lambda_F(\frac{DK_m}{NRB_r}, B_r), \end{aligned} \tag{3.1}$$

where $\mathbb{1}_{[\cdot]}$ is an indicator function, and $\lambda_F(\cdot)$ is defined to be:

$$\lambda_F(n, b) = \left(\frac{1}{2F(F-1)} n^2 + \frac{3}{2} n - \frac{F^2}{2(F-1)} \right) \cdot b. \tag{3.2}$$

Proof. Our analysis includes five I/O-types listed in Table 3.3. Each map task reads a data chunk of size C as input, and writes $C \cdot K_m$ bytes as output. Given the workload D, we have

D/C map tasks in total and $D/(C \cdot N)$ map tasks per node. So, the input cost, U_1, and output cost, U_3, of all map tasks on a node are:

$$U_1 = \frac{D}{N} \quad \text{and} \quad U_3 = \frac{D \cdot K_m}{N}.$$

The size of the reduce output on each node is $U_5 = \frac{D \cdot K_m \cdot K_r}{N}$.

Map and reduce internal spills result from the multi-pass merge operation, which can take place in a map task if the map output exceeds the memory size and hence needs to use external sorting, or in a reduce task if the reduce input data does not fit in memory.

We make a general analysis of multi-pass merge first. Suppose that our task is to merge n sorted runs, each of size b. As these initial sorted runs are generated, they are written to spill files on disk as $f_1, f_2, \ldots$ Whenever the number of files on disk reaches $2F - 1$, a background thread merges the *smallest* F files into a new file on disk. We label the new merged files as $m_1, m_2, \ldots$ Figure 3.5 illustrates this process, where an unshaded box denotes an initial spill file and a shaded box denotes a merged file. For example, after the first $2F - 1$ initial runs generated, $f_1, \ldots, f_F$ are merged together and the resulting files on disk are $m_1, f_{F+1}, \ldots, f_{2F-1}$ in order of decreasing size. Similarly, after the first $F^2 + F - 1$ initial runs are generated, the files on disk are $m_1, \ldots, m_F, f_{F^2+1}, \ldots, f_{F^2+F-1}$. Among them, $m_1, f_{F^2+1}, \ldots, f_{F^2+F-1}$ will be merged together and the resulting files on disk will be $m_{F+1}, m_2, \ldots, m_F$ in order of decreasing size. After all initial runs are merged, a final merge combines all the remaining files (there are at most $2F - 1$ of them).

For the analysis, let α_i denote the size of a merged file on level i ($2 \le i \le h$) and let $\alpha_1 = b$. Then $\alpha_i = \alpha_{i-1} + (F-1)b$. Solving this recursively gives $\alpha_i = (i-1)Fb - (i-2)b$. Hence, the total size of all the files in the first h levels is:

$$F(\alpha_h + \sum_{i=1}^{h-1} (\alpha_i + (F-1)b)) = bF\left(hF + \frac{(F-1)(h-2)(h+1)}{2}\right).$$

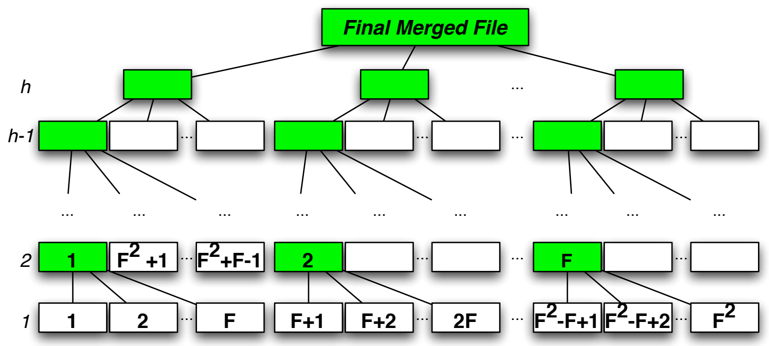


Figure 3.5. Analysis of the tree of files created in multi-pass merge.

If we count all the spill files (unshaded boxes) in the tree, we have $n = (F + (F-1)(h-2))F$. Then we substitute h with n and F using the above formula and get

$$\lambda_F(n,b) = \left(\frac{1}{2F(F-1)}n^2 + \frac{3}{2}n - \frac{F^2}{2(F-1)}\right) \cdot b$$

Then, the total I/O cost is $2\lambda_F(n,b)$ as each file is written once and read once. The remaining issue is to derive the exact numbers for n and b in the multi-pass merge in a map or reduce task.

In a map task, if its output fits in the map buffer, then the merge operation is not needed. Otherwise, we use the available memory to produce sorted runs of size B_m each and later merge them back. So, $b = B_m$ and $n = \frac{C \cdot K_m}{B_m}$. As each node handles $D/(C \cdot N)$ map tasks, we have the I/O cost for map internal spills on this node as:

$$U_2 = \begin{cases} \frac{2D}{C \cdot N} \cdot \lambda_F(\frac{C \cdot K_m}{B_m}, B_m) & \text{if } C \cdot K_m > B_m; \\ 0 & \text{otherwise.} \end{cases}$$

In a reduce task, as we do not have a combine function, the input for reduce usually cannot fit in memory. The size of input to each reduce task is $\frac{D \cdot K_m}{N \cdot R}$. So, $b = B_r$ and $n = \frac{D \cdot K_m}{N \cdot R \cdot B_r}$. As each node handles R reduce tasks, we have the reduce internal spill cost:

$$U_4 = 2R \cdot \lambda_F(\frac{D \cdot K_m}{N \cdot R \cdot B_r}, B_r)$$

Summing up $U_1, \ldots, U_5$, we then have Equation 3.1 in the proposition. $\square$

3.3.1.2 Modeling the Number of I/O requests

In our analysis we also model the number of I/O requests in a Hadoop job, which allows us to estimate the disk seek time when these I/O requests are performed as random I/O operations. Again we summarize our result in the following proposition.

Proposition 3.3.2. *Given the workload description (D, K_m, K_r) and the hardware description (N, B_m, B_r), as defined in Table 3.3, the number of I/O requests in a Hadoop job is:*

$$\begin{aligned} S =& \frac{D}{CN}\left(\alpha + 1 + \mathbb{1}_{[CK_m > B_m]} \cdot \left(\lambda_F(\alpha, 1)(\sqrt{F}+1)^2 + \alpha - 1\right)\right) \\ &+ R\left(\beta K_r(\sqrt{F}+1) - \beta\sqrt{F} + \lambda_F(\beta, 1)(\sqrt{F}+1)^2\right), \end{aligned}$$

where $\alpha = \frac{CK_m}{B_m}$, $\beta = \frac{DK_m}{NRB_r}$, $\lambda_F(\cdot)$ is defined in Equation 3.2, and $\mathbb{1}_{[\cdot]}$ is an indicator function.

Proof. We again consider the five types of I/O listed in Table 3.3. For each map task, a chunk of input data is sequentially read until the map output buffer fills up or the chunk is completely finished. So, the number of I/O requests for the map input $\frac{C \cdot K_m}{B_m}$. All map tasks on a node will trigger the number of I/O requests, S_1, as:

$$S_1 = (\frac{D}{CN}) \cdot (\frac{CK_m}{B_m}).$$

If the map output fits in memory, there is no internal spill and the map output is written to disk using one sequential I/O. Considering all map tasks on a node, we have

$$S_2 + S_3 = \frac{D}{CN} \text{ if } CK_m \le B_m .$$

If the map output exceeds the memory size, it is sorted using external sorting which involves multi-pass merge.

Since both map and reduce tasks may involve multi-pass merge, we first do a general analysis of the I/O requests incurred in this process. How many I/O requests to make depends not only on the data size but also on the memory allocation scheme, which can vary with the implementation and system resources available. Hence, we consider the optimal scheme regarding the I/O requests below.

Suppose that a merge step is to merge F files, each of size f, into a new file with memory size B. For simplicity, we assume the buffer size for each input file is the the same, denoted by B_{in}. Then the buffer size for the output file is $B - F \cdot B_{in}$. The number of read and write requests is

$$s = \frac{F \cdot f}{B_{in}} + \frac{F \cdot f}{B - F \cdot B_{in}}.$$

By taking the derivative with respect to B_{in} we can minimize s, which is:

$$s^{opt} = \frac{F \cdot f}{B}(\sqrt{F}+1)^2 \quad \text{when} \quad B_{in}^{opt} = \frac{B}{F+\sqrt{F}}.$$

Revisit the tree of files in multi-pass merge in Figure 3.5. Each merge step, numbered j in the formula below, corresponds to the creation of a merged file (shaded box) in the tree. When we sum up the I/O requests of all these steps, we can apply our previous result on the total size of all the files:

$$\sum_j s_j^{opt} = \frac{\sum_j F \cdot f_j}{B}(\sqrt{F}+1)^2 = \frac{\lambda_F(n,b)}{B}(\sqrt{F}+1)^2,$$

where n is the number of initial spill files containing sorted runs and b is the size of each sorted run. But this above analysis does not include the I/O requests of writing the n initial sorted runs from memory to disk, so we add n requests and have the total number:

$$s^{merge} = n + \frac{\lambda_F(n,b)}{B}(\sqrt{F}+1)^2. \tag{3.3}$$

The value of n and b in map and reduce tasks have been analyzed previously. In a map task, if $CK_m > B_m$, then multi-pass merge takes place. For the above formula, $B = B_m$, $b = B_m$ and $n = \frac{CK_m}{B_m}$. Considering all $\frac{D}{CN}$ map tasks on a node, we have:

$$S_2 + S_3 = \frac{D}{CN} \left(\frac{CK_m}{B_m} + \lambda_F(\frac{CK_m}{B_m}, 1)(\sqrt{F} + 1)^2 \right) \quad \text{if} \quad CK_m > B_m.$$

For a reduce task, we have $B = B_r$, $b = B_r$ and $n = \frac{DK_m}{NRB_r}$. We can get the I/O requests by plugging these values in Equation 3.3. However, this result includes the disk requests for writing output in the final merge step, which does not actually exist because the output of the final merge is directly fed to the reduce function. The overestimation is the number of requests for writing data of size $\frac{DK_m}{NR}$ with an output buffer of size $B_r - F \cdot B_{in}^{opt} = \frac{B_r}{\sqrt{F}+1}$. So, the overestimated number of requests is $\frac{DK_m(\sqrt{F}+1)}{NRB_r}$. Given R reduce tasks per node, we have:

$$S_4 = R \left(\lambda_F(\frac{DK_m}{NRB_r}, 1)(\sqrt{F} + 1)^2 - \frac{DK_m}{NRB_r} \cdot \sqrt{F} \right).$$

Finally, the output size of a reducer task is $\frac{DK_mK_r}{NR}$, written to disk with an output buffer of size $\frac{B_r}{\sqrt{F}+1}$. So, we can estimate the I/O requests for all reduce tasks on a node, S_5, with

$$S_5 = R \left(\frac{DK_m}{NRB_r} \cdot K_r(\sqrt{F} + 1) \right).$$

The sum of $S_1, \ldots, S_5$ gives the result in the proposition. $\square$

We note that for common workloads, the I/O cost is dominated by the cost of reading and writing all the bytes, not the seek time. We provide detailed empirical evidence in Section 3.3.2.

3.3.1.3 Modeling the Startup Cost

We further consider the cost of starting map and reduce tasks as it has been reported to be a nontrivial cost [76]. Since the number of map tasks is usually much larger than that of reduce tasks, we mainly consider the startup cost for map tasks. If c_m is the cost in second of creating a map task, the total map startup cost per node is $c_{start} \cdot \frac{D}{CN}$.

3.3.1.4 Combining All in Time Measurement

Let U be the number of bytes read and written in a Hadoop job and let S be the number of I/O requests made. Let c_{byte} denote the sequential I/O time per byte and c_{seek} denote the disk seek time for each I/O request. We define the time measurement T that combines the cost of reading and writing all the bytes, the seek cost of all I/O requests, and the map startup cost as follows:

$$T = c_{byte} \cdot U + c_{seek} \cdot S + c_{start} \cdot \frac{D}{CN}. \tag{3.4}$$

The above formula is our complete analytical model that captures the effects of all of the involved parameters.

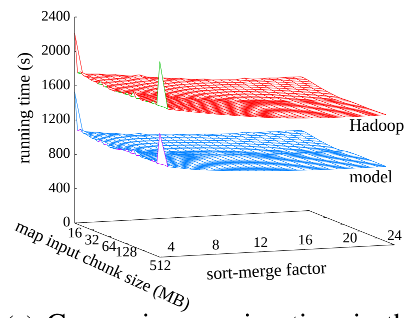
3.3.2 Optimizing Hadoop based on our Analytical Model

Our analytical model enables us to predict system behaviors as Hadoop parameters vary. Then, given a workload and system configuration, we can choose values of these parameters that minimize the time cost in our model, thereby optimizing Hadoop performance.

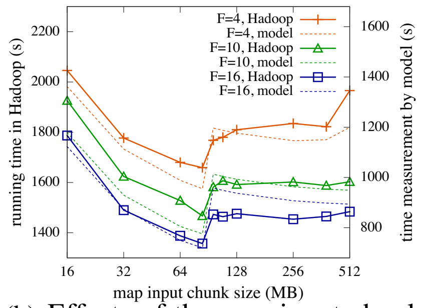
3.3.2.1 Optimizations

To show the effectiveness of our model, we compare the predicted system behavior based on our model and the actual running time measured in our Hadoop cluster. We used the sessionization task and configured the workload, our cluster, and Hadoop as follows: (1) Workload: D=97GB, K_m=K_r=1;[3] (2) Hardware: N=10, B_m=140MB, B_r=260MB; (3) Hadoop: R=4 or 8, and varied values of C and F. We also fed these parameter values to our

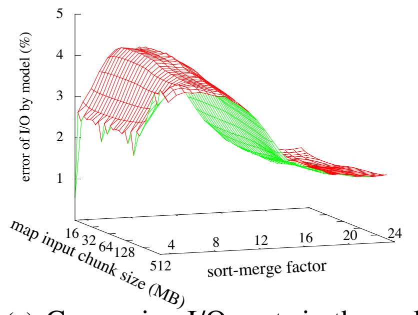
[3]We used a smaller dataset in this set of experiments compared to the benchmark because changing Hadoop configurations often required reloading data into HDFS, which was very time-consuming.



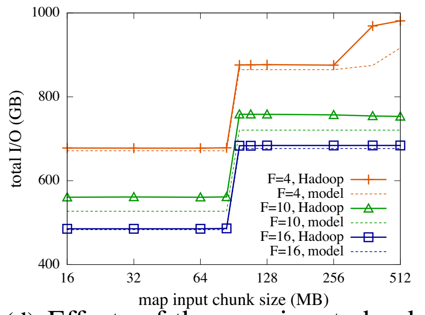
(a) Comparing running time in the real system and time measurement in our model



(b) Effects of the map input chunk size C and the merge factor F on time measurements



(c) Comparing I/O costs in the real system and in our model



(d) Effects of the map input chunk size C and the merge factor F on the I/O cost

Figure 3.6. Validating our model against actual measurements in a Hadoop cluster.

analytical model. In addition, we set the constants in our model by assuming sequential disk access speed to be 80MB/s, disk seek time to be 4 ms, and the map task startup cost to be 100 ms.

Our first goal is to validate our model. In our experiment, we varied the map input chunk size, C, and the the merge factor, F. Under 100 different combinations of (C, F), we measured the running time in a real Hadoop system, and calculated the time cost predicted by our model. The result is shown as a 3-D plot in Figure 3.6(a).[4] Note that our goal is not to compare the absolute values of these two time measurements: In fact, they are not directly comparable, as the former is simply a linear combination of the startup cost and I/O

[4]For both the real running time and modeled time cost, the respective 100 data points were interpolated into a finer-grained mesh.

costs based on our model, whereas the latter is the actual running time affected by many system factors as stated above. Instead, we expect our model to predict the changes of the time measurement when parameters are tuned, so as to identify the optimal parameter setting. Figure 3.6(a) shows that indeed the performance predicted by our model and the actual running time exhibit very similar trends as the parameters C and F are varied. We also compared the I/O costs predicated by our model and those actually observed. Not only do we see matching trends, the predicted numbers are also close to the actual numbers. As shown in Figure 3.6(c), the differences between the predicted numbers and the actual numbers are mostly within 5%. Here the errors are mainly due to the fact that our analysis assumes the multi-pass merge tree to be full but this is not always true in practice.

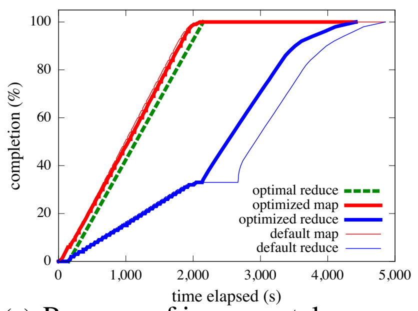
Our next goal is to show how to optimize the parameters based on our model. To reveal more details from the 3-D plots, we show the results of a smaller range of (C, F) in Figure 3.6(b) and Figure 3.6(d) where the solid lines are for the actual measurements from the Hadoop cluster and the dashed lines are for predication using our model.

Optimizing the Chunk Size. When the chunk size C is very small, the MapReduce job uses many map tasks and the map startup cost dominates in the total time cost. As C increases, the map startup cost reduces. However, once the map output exceeds its buffer size, multi-pass merge is incurred with increased I/O cost, as shown in Figure 3.6(d). As a result, the time cost jumps up at this point, and then remains nearly constant since the reduction of startup cost is not significant. When C exceeds a large size (whose exact value depends on the merge factor, e.g., size 256 when F=4 shown in Figure 3.6(d)), the number of passes of on-disk merge goes up, thus incurring more I/Os. The overall best performance in running time is observed at the maximum value of C that allows the map output to fit in the buffer. Given a particular workload, we can easily estimate K_m, the ratio of output size to input size, for the map function and estimate the map output buffer size B_m to be about $\frac{2}{3}$ of the total map memory size (given the use of other metadata). Then we can choose the maximum C such that $C \cdot K_m \leq B_m$.

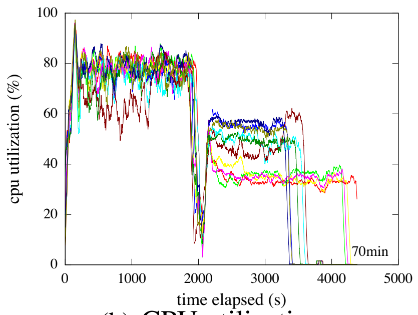
Optimizing the Merge Factor. We then investigate the merge factor, F, that controls how frequently on-disk files are merged in the multi-pass merge phase. Figure 3.6(b) shows three curves for three F values. The time cost decreases with larger values of F (from 4 to 16), mainly due to fewer I/O bytes incurred in the multi-pass merge as shown in Figure 3.6(d). When F goes up to the number of initial sorted runs (around 16), the time cost does not decrease further because all the runs are merged in a single pass. For several other workloads tested, one-pass merge was also observed to provide the best performance.

Our model can also reveal potential benefits of small F values. When F is small, the number of files to merge in each step is small, so the reads of the input files and the writes of the output file are mostly sequential I/O. As such, a smaller F value incurs more I/O bytes, but fewer disk seeks. According to our model, the benefits of small F values can be shown only when the system is given limited memory but a very large data set, e.g., several terabytes per node, which is beyond the current storage capacity of our cluster.

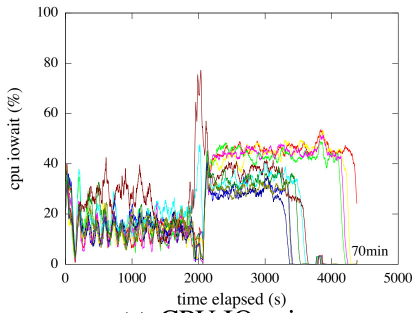
Effect of the Number of Reducers. The third relevant parameter is the number of reducers per node, R. The original MapReduce proposal [25] has recommended R to be the number of cores per node times a small constant (e.g., 1 or 2). As this parameter does not change the workload but only distributes it over a variable number of reduce workers, our model shows little difference as R varies. Empirically, we varied R from 4 to 8 (given 4 cores on each node) while configuring C and F using the most appropriate values as reported above. Interestingly, the run with R=4 took 4,187 seconds, whereas the run with R=8 took 4,723 seconds. The reasons are two-fold. First, by tuning the merge factor, F, we have minimized the work in multi-pass merge. Second, given 4 cores on each node, we have only 4 reduce task slots per node. Then for R=8, the reducers are started in two waves. In the first wave, 4 reducers are started. As some of these reducers finish, a reducer in the second wave can be started. As a consequence, the reducers in the first wave can read map output soon after their map tasks finish, hence directly from the local memory. In contrast, the reducers in the second wave are started long after the mappers have finished. So they have to fetch



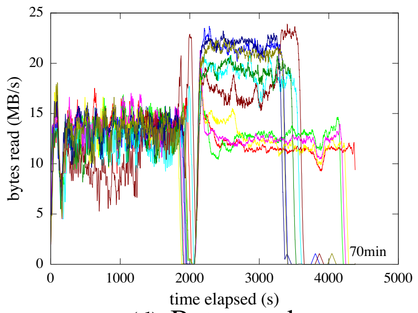
(a) Progress of incremental processing



(b) CPU utilization



(c) CPU IOwait



(d) Bytes read

Figure 3.7. Performance of optimized Hadoop based on our model.

map output from disks, hence incurring high I/O costs in shuffling. Our conclusion is that optimizing the merge factor, F, can reduce the actual I/O cost in multi-pass merge, and is a more effective method than enlarging the number of reducers beyond the number of reduce task slots available at each node.

In summary, the above results demonstrate two key benefits of our model: (1) Our model predicts the trends in I/O cost and time cost close to the observations in real cluster computing. (2) Given a particular workload and hardware configuration, one can run our model to find the optimal values of the chunk size C and merge factor F, and choose an appropriate value of R based on the recommendation above.

3.3.2.2 Analysis of Optimized Hadoop

We finally reran the 240GB sessionization workload described in our benchmark (see Section 3.1). We optimized Hadoop using 64MB data chunks, one-pass merge, and 4 reducers per node as suggested by the above results. The total running time was reduced from 4,860 seconds to 4,187 seconds, a 14% reduction of the total running time.

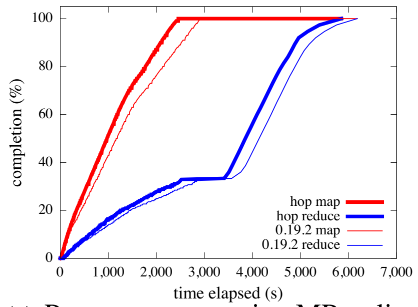
Given our goal of delivering a query answer as soon as all relevant data has arrived, a key requirement is to perform *incremental processing*. In this regard, we propose metrics for the map and reduce progress, as defined below.

Definition 1 (Incremental Map and Reduce Progress)**.** *The map progress is defined to be the percentage of map tasks that have completed. The reduce progress is defined to be:* $\frac{1}{3}\cdot$ *% of shuffle tasks completed +* $\frac{1}{3}\cdot$ *% of combine function or reduce function completed +* $\frac{1}{3}\cdot$ *% of reduce output produced.*

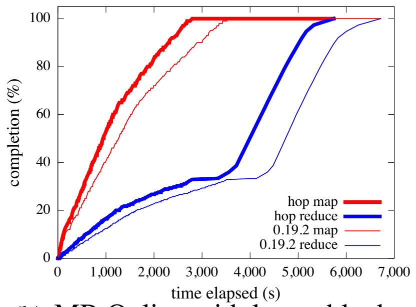
Note that our definition differs from the default Hadoop progress metric where the reduce progress includes the work on multi-pass merge. In contrast, we discount multi-pass merge because it is irrelevant to a user query, and emphasize the actual work on the reduce function or combine function and the output of answers.

Figure 3.7(a) shows the progress of optimized Hadoop in bold lines (and the progress of stock Hadoop in thin lines as a reference). The map progress increases steadily and reaches 100% around 2,000 seconds. The reduce progress increases to 33% in these 2,000 seconds, mainly because the shuffle progress could keep up with the map progress. Then the reduce progress slows down, due to the overhead of merging, and lags far behind the map progress. The optimal reduce progress, as marked by a dashed line in this plot, keeps up with the map progress, thereby realizing fast incremental processing. As can be seen, there is a big gap between the optimal reduce progress and what the optimized Hadoop can currently achieve.

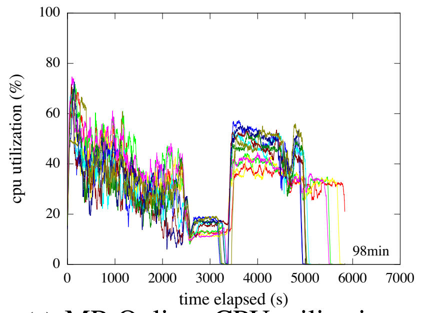
Figure 3.7(b), 3.7(c), and 3.7(d) further show the CPU utilization, CPU iowait, and the number of bytes read using optimized Hadoop. We make two main observations: (1) The CPU utilization exhibits a smaller dip in the middle of a job compared to stock Hadoop in Figure 3.2(b). However, the CPU cycles consumed by the mappers, shown as the area under the curves before 2,000 seconds, are about the same as those using stock Hadoop. Hence, the CPU overhead due to sorting, as mentioned in our benchmark, still exists. (2) The CPU



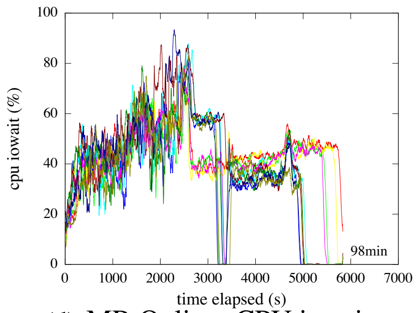
(a) Progress report using MR online.



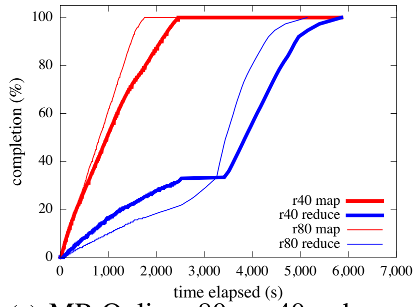
(b) MR Online with larger blocks.



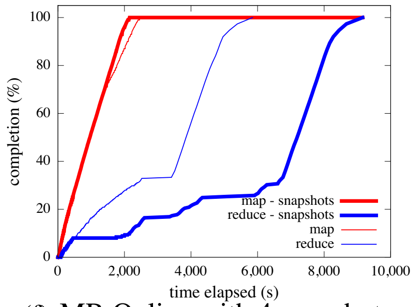
(c) MR Online: CPU utilization.



(d) MR Online: CPU iowait.



(e) MR Online: 80 vs. 40 reducers.



(f) MR Online with 4 snapshots.

Figure 3.8. Performance of MapReduce Online with pipelining of data.

iowait plot still shows a spike in the middle of job and remains high in the rest of the job. This is due to the blocking of CPU by the I/O operations in the remaining single-pass merge.

3.3.3 Pipelining in Hadoop

Another attempt to optimize Hadoop for incremental processing would be to pipeline data from mappers to reducers so that reducers can start the work earlier. This idea has been implemented in MapReduce Online [22], as described in Section 3.2.4. In our benchmark, we made the following observations about pipelining data from mappers to reducers in the Hadoop framework:

(1) Benefits of Pipelining. We first summarize the benefits of pipelining that were observed in our benchmark. First, pipelining data from mappers to reducers can result in small performance benefits. For instance, for sessionization, Figure 3.8(a) shows 5% improvement in total running time over the version of stock Hadoop, 0.19.2, on which MapReduce Online's code is based. However, the overall performance gain of MapReduce Online over Hadoop is small (e.g., 5%), less that the gain of our model-based optimization (e.g., 14%). Second, pipelining also makes the performance less sensitive to the HDFS chunk size. As Figure 3.8(b) shows, when the chunk size is increased from 64MB to 256MB, the performance of MapReduce Online, denoted by the bold lines, stays about the same as that of 64MB, whereas the performance of stock Hadoop, denoted by the thin lines, degrades. This is because a larger block size places additional strain on the map output buffer and therefore increases the chance of having to spill the map output to disk in order to perform sorting. Pipelining, however, is able to mitigate this effect by eagerly sending data at a finer granularity to the reducers with the intention to use merging later to bring all the data in sorted order.

(2) Limitations of Pipelining. However, we observe that adding pipelining to an overall blocking implementation based on sort-merge is not an effective mechanism for incremental processing. Most importantly, the reduce progress lags far behind the map progress, as shown in Figure 3.8(a) and 3.8(b). To explain this behavior, we observe from Figure 3.8(c) that the CPU utilization still has low values in the middle of the job and is on the average 50% or below over the entire job. While CPU can be idle due to I/O wait or network wait (given the different communication model used), the CPU iowait in Figure 3.8(d) again shows a spike in the middle of the job and overall high values during the job. Hence, the problems with blocking and intensive I/O due to multi-pass merge still exist.

(3) Effect of Adding Reducers. We further investigate whether using more reducers can help close the gap between the map progress and reduce progress. It is important to note that pipelining does not reduce the total amount of work in sort-merge but rather rebalances the work between the mappers and reducers. More specifically, eager transmission of data from mappers to reducers reduces the sorting work in the mappers but increases the merge work in the reducers. To handle more merge work, increasing the number of reducers helps improve the overall running time, as shown in Figure 3.8(e) where the number of reducers is increased from 4 per node to 8 per node. However, the reduction of the running time comes at the cost of significantly increased resource consumption and the gap between the map progress and the reduce progress is not reduced much. Moreover, further increasing the number of reducers, e.g., to 12 per node, starts to perform worse due to the drawbacks of using multiple waves of reducers mentioned in Section 3.3.2.1.

(4) Effect of using Snapshots. Finally, MapReduce Online has an extension to periodically output snapshots (e.g., when reducers have received 25%, 50%, 75%, ..., of the data). However, this is done by repeating the merge operation for each snapshot, not by incremental in-memory processing. As a result, the simple snapshot-based mechanism can incur high I/O overheads and significantly increased running time, as shown in Figure 3.8(f).

Summary We close the discussion in this section with the summary below:

- ▶ Our analytical model can be used to choose appropriate values of Hadoop parameters, thereby improving performance.
- ▶ Optimized Hadoop, however, still has a significant barrier to fast incremental processing: (1) The remaining one-pass merge can still incur blocking and a substantial I/O cost. (2) Due to the above reason, the reduce progress falls far behind the map progress. (3) The map tasks still have the high CPU cost of sorting.
- ▶ Pipelining from mappers to reducers does not resolve the blocking or I/O overhead in Hadoop, hence not an effective mechanism for providing fast incremental processing.

3.4 Summary

In this chapter, we examined whether the existing MapReduce implementations can support incremental processing by tuning system parameters, and the architectural design changes that are necessary to bring the benefits of the MapReduce model to incremental processing. Our empirical and theoretical analysis showed that the widely-used sort-merge implementation for MapReduce data parallelism poses a fundamental barrier to incremental analytics, despite optimizations.

CHAPTER 4

INCREMENTAL PROCESSING

Based on the insight from the last chapter that the sort-merge implementation in the original MapReduce model poses a fundamental barrier to incremental processing, we next answer the question: "What are the necessary architecture changes in MapReduce in order to support incremental processing?" Incremental processing means that computation is performed as soon as the relevant data becomes available. In this chapter, we propose a new data analysis platform based on MapReduce that is geared for incremental processing. We first consider a finite input from disk in this chapter and will extend to stream inputs in the next chapter. We made two key architecture changes to Hadoop:

Our first mechanism replaces the sort-merge implementation in Hadoop with a purely hash-based framework, which is designed to address the computational and I/O bottlenecks as well as the blocking behavior of sort-merge. We devise two hash techniques to suit different reduce functions, depending on whether the reduce function permits incremental processing or not. Besides eliminating the sorting cost from the map tasks, these hash techniques can provide fast in-memory processing of the reduce function when the memory reaches a sufficient size as determined by the workload and algorithm.

Our second mechanism further brings the benefits of fast in-memory processing to workloads that require a large key-state space that far exceeds available memory. We propose both deterministic and randomized techniques to dynamically recognize popular keys and then update their states using a full in-memory processing path, both saving I/Os and enabling early answers for these keys. Less popular keys trigger I/Os to stage data to disk but have limited impact on the overall efficiency.

Experiments on a range of workloads in click stream analysis and web document analysis show the following main results: (1) Our hash techniques significantly improve the progress of the map tasks, due to the elimination of sorting, and given sufficient memory, enable fast in-memory processing of the reduce function. (2) For challenging workloads that require a large key-state space, our dynamic hashing mechanism significantly reduces I/Os and enables the reduce progress to keep up with the map progress, thereby realizing incremental processing. For instance, for sessionization over a click stream, the reducers output user sessions as data is read and finish as soon as all mappers finish reading the data in 34.5 minutes, triggering only 0.1GB internal data spill to disk in the job. In contrast, the original Hadoop system returns all the results towards the end of the 81 minute job, writing 370GB internal data spill to disk. (3) Further trade-offs exist between our hash-based techniques under different workload types, data localities, and memory sizes, with dynamic hashing working the best under constrained memory and most workloads.

This chapter is organized as follows. We first show the hash-based techniques in Section 4.1. We then describe our prototype implementation in Section 4.2. Finally, we present the evaluation results in Section 4.3, and conclude in Section 4.5.

4.1 A New Hash-based Platform

To build the new platform, our first mechanism is to devise a hash-based alternative to the widely used sort-merge implementation for data parallelism, with the goal to minimize computational and I/O bottlenecks as well as blocking. The hash implementation can be particularly useful when analytical tasks do not require the output of the reduce function to be sorted across different keys.[1] More specifically, we design two hash techniques for

[1]Our implementation offers a knob for a MapReduce job to be configured with either the hash implementation or the sort-merge implementation. When our platform is used to build a query processor on top of MapReduce, if both sort-merge and hashing algorithms are available for implementing an operator like join, our system will enable the query processor to quickly implement the hash algorithm of choice by utilizing the internal hashing functionality of our MapReduce platform.

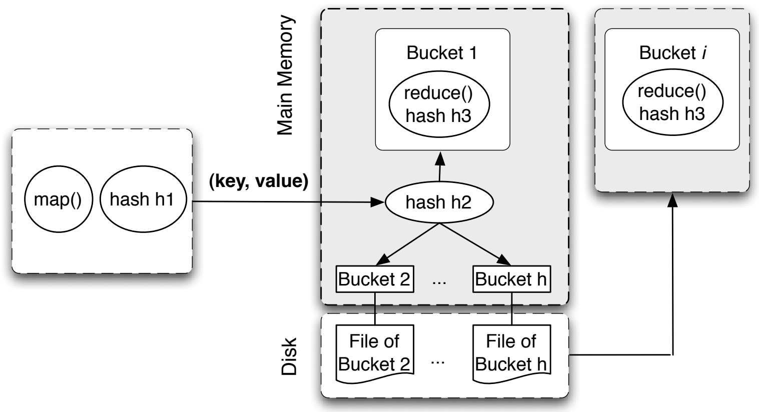


Figure 4.1. MR-hash: hashing in mappers and two phase hash processing in reducers.

two different types of reduce functions, respectively, which we describe in Section 4.1.1 and Section 4.1.2, respectively. These techniques enable fast in-memory processing when there is sufficient memory for a given workload. In addition, our second mechanism brings the benefits of such fast in-memory processing further to workloads that require a large key-state space far exceeding available memory. Our techniques dynamically identify popular keys and update their states using a full in-memory processing path. These dynamic techniques are described in Section 4.1.3. In Section 4.1.4, we discuss the optimization of key Hadoop system parameters in our hash-based framework.

4.1.1 A Basic Hash Technique (MR-hash)

Recall from Section 3.1 that to support parallel processing, the MapReduce computation model implements "*group data by key, then apply the reduce function to each group*". The main idea underlying our hash framework is to implement the MapReduce group-by functionality using a series of independent hash functions h_1, h_2, h_3, . . ., across the mappers and reducers.

As depicted in Figure 4.1, the hash function h_1 partitions the map output into subsets corresponding to the scheduled reducers. Hash functions h_2, h_3, . . ., are used to implement group-by at each reducer. We adopt the hybrid-hash algorithm [85] from parallel databases as follows: h_2 partitions the input data to a reducer to n buckets, where the first bucket, say, D_1, is held completely in memory and other buckets are streamed out to disks as their write buffers fill up. This way, we can perform group-by on D_1 using the hash function h_3 and apply the reduce function to each group in memory. Other buckets are processed subsequently, one at a time, by reading the data from the disk. If a bucket D_i fits in memory, we use in-memory processing for the group-by and the reduce function. Otherwise, we recursively partition D_i using hash function h_4, and so on. In our implementation, we use standard universal hashing to construct a series of independent hash functions.

Following the analysis of the hybrid hash join [85], simple calculation shows that if h_2 can evenly distribute the data into buckets, recursive partitioning is not needed if the memory size is greater than $2\sqrt{|D|}$, where $|D|$ is the number of pages of data sent to the reducer, and the I/O cost is $2(|D| - |D_1|)$ pages of data read and written. The number of buckets, h, can be derived from the standard analysis by solving a quadratic equation.

The above technique, called MR-hash, exactly matches the current MapReduce model that collects all the values of the same key into a list and feeds the entire list to the reduce function. This baseline technique in our work is similar to the hash technique used in parallel databases [29], but implemented in the MapReduce context. Compared to stock Hadoop, MR-hash offers several benefits: First, on the mapper side, it avoids the CPU cost of sorting as in the sort-merge implementation. Second, this hash implementation offers a step towards incremental processing: It allows answers for the first bucket, S_0, to be returned from memory after all the data arrives, and answers for other buckets to be returned one bucket at a time. In contrast, sort-merge cannot return any answer until all the data is sorted. However, such incremental processing is very coarse-grained, as a bucket can contain a large chunk of data. Moreover, in many cases the total I/O is not significantly better than the I/O of sort-merge [80].

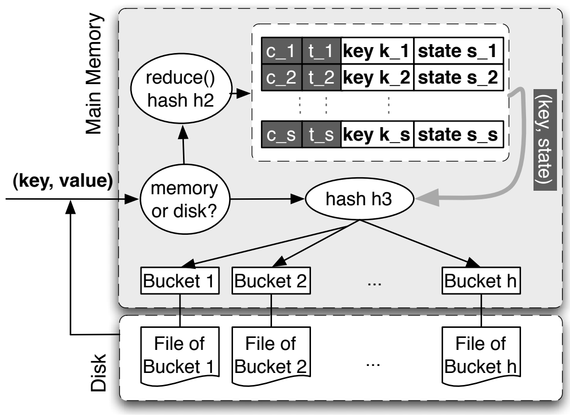


Figure 4.2. (Dynamic) Incremental Hash: monitoring keys and updating states.

4.1.2 An Incremental Hash Technique (INC-hash)

Our second hash technique is designed for reduce functions that permit incremental processing, including simple aggregates like *sum* and *count*, and more complex problems that have been studied in the area of sublinear-space stream algorithms [71]. For such reduce functions, we propose a more efficient hash algorithm, called incremental hash (INC-hash).

Algorithm The algorithm is illustrated in Figure 4.2 (the reader can ignore the darkened boxes for now, as they are used only in the third technique). In Phase 1, as a reducer receives map output, called tuples for simplicity, we build a hash table H (using hash function h_2) that maps from a key to the *state* of computation for all the tuples of that key that have been seen so far. When a new tuple arrives, if its key already exists in H, we update the key's state using the new tuple. If its key does not exist in H, we add a new key-state pair to H if there is still memory. Otherwise, we hash the tuple (using h_3) to a bucket, place the tuple in the write buffer, and flush the write buffer when it becomes full (similar to Hybrid Cache [39] in this step). At the end of Phase 1, the reducer has seen all the tuples and returned final answers for all the keys in H. Then in Phase 2, it reads disk-resident buckets back one at a time, repeating the procedure above to process each bucket. If the key-state pairs produced from a specific bucket fit in memory, no further I/O will be incurred.

Otherwise, the algorithm will again process some keys in memory and write the tuples of other keys to disk-resident buckets, i.e., applying recursive hashing.

INC-hash offers two major advantages over MR-hash: (1) *Reduced data volume and I/O*: For those keys held in memory, their tuples are continuously collapsed into states in memory, hence avoiding I/O's for those tuples altogether. I/O's can be completely avoided in INC-hash if the memory is large enough to hold all *key-state* pairs, in contrast to all the data in MR-hash. (2) *Earlier results*: For those keys held in memory, query answers can be returned before all the data is seen. In particular, earlier results are possible for filter queries (e.g., when the count of a URL exceeds a threshold), join queries (whose results can be pipelined out), and window queries (whose results can be output whenever a window closes).

Partial Aggregation An opportunity for further optimization is that some reduce functions that permit incremental processing are also amenable to partial aggregation, which splits incremental processing to a series of steps on the processing pipeline with successively reduced data volume. Classical examples are the aggregates sum, $count$, and avg. To support partial aggregation in the MapReduce context, we define three primitive functions:

- ▶ The initialize function, $init()$, reduces a sequence of data items of the same key to a state;
- ▶ The combine function, $cb()$, reduces a sequence of states of the same key to a new state;
- ▶ The finalize function, $fn()$, produces a final answer from a state.

The initialize function is applied immediately when the map function finishes processing. This changes the data in subsequent processing from the original key-value pairs to key-state pairs. The combine function can be applied to any intermediate step that collects a set of states for the same key, e.g., in updating a key-state pair in the hash table with a new data item (which is also a key-state pair) or in packing multiple data items in the write buffer of a bucket. Finally, the original reduce function is implemented by $cb()$ followed by $fn()$.

Partial aggregation provides several obvious benefits: The initialize function reduces the amount of data output from the mappers, thereby reducing the communication cost in shuffling and the CPU and I/O costs of subsequent processing at the reducers. In addition, the reducers can apply $cb()$ in all suitable places to collapse data aggressively into compact states, hence reducing the I/O cost.

In implementation, the INC-hash algorithm is applied to "group data by key" in both $init()$ and $cb()$. The operation within each group, in both $init()$ and $cb()$, is very similar to the user-specified reduce function, as in the original proposal of combiner functions [25].

Memory and I/O Analysis We next analyze the INC-hash algorithm for its memory requirements and I/O cost. Let D be the size of data input to the algorithm (e.g., data sent to a reducer), U be the size of the key-state space produced from the data, and B be the memory size, all in terms of the number of pages covered. Let h be the number of buckets created in the INC-hash algorithm. In Phase 1, we need 1 page for the input data and h pages for write buffers of the disk-resident buckets. So, the size of the hash table is $B - h - 1 \geq 0$. Assume that the key-state pairs not covered by the hash table, whose size is $U - (B - h - 1)$, are evenly distributed across h buckets. To make sure that the key-state pairs produced from each bucket fit in memory in Phase 2, the following equality has to hold: $U - (B - h - 1) \leq h \cdot (B - 1)$. Rewriting both constraints, we have:

$$\frac{U-1}{B-2} - 1 \leq h \leq B - 1. \tag{4.1}$$

The above analysis of memory requirements has several implications:

- When the memory size B reaches U+1, all the data is processed in memory, i.e., $h = 0$.

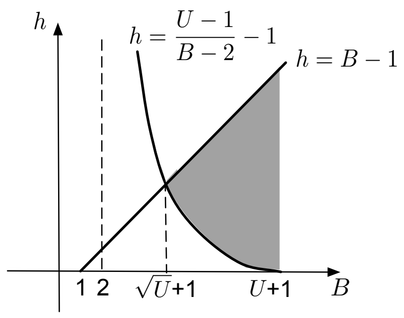


Figure 4.3. Determining the number of buckets h given memory size B and key-state space size U.

- When B is in the range $[\sqrt{U} + 1, U]$, given a particular value of B, the number of buckets h can be chosen between the lower bound $(U - 1)/(B - 2) - 1$ and the upper bound $B - 1$, as marked by the shaded area in Figure 4.3.
- Also for the above range of B, under the assumption of uniform distribution of keys in the data, no recursive partitioning is needed in INC-hash: those tuples that belong to the in-memory hash table are simply collapsed into the states, and other tuples are written out and read back exactly once. In this case, the fraction of the data that is not collapsed into the in-memory hash table is $(U - (B - h - 1))/U$. So the I/O cost of INC-hash is:
$$2 \cdot (1 - \frac{B - h - 1}{U}) \cdot D. \tag{4.2}$$
- To minimize I/O according to the above formula, we want to set the number of buckets h to be its lower bound $(U - 1)/(B - 2) - 1$.

We note that for common workloads, the memory size is expected to exceed $\sqrt{U}$. All modern computers with several GB's of memory can meet this requirement for any practical value of U.

Sensitivity to Parameters The above analysis reveals that the optimal I/O performance of INC-hash requires setting the right number of buckets, h, which depends on the size of the key-state space, U. This issue is less a concern in traditional databases because the

DBMS can collect detailed statistics from stored data and size estimation for relational operators has been well studied. For our problem of scalable, incremental analytics, the lack of knowledge of the key space often arises when data is streamed over the wire or from upstream complex operators (coded in user-defined functions) in the processing pipeline. Consider the performance loss when we do not know the key space size. According to Equation 4.1, without knowing U we do not know the lower bound of h and hence the only safe value to choose would be the upper bound $B-1$. Given this worst choice of h, we pay the following extra I/O according to Equation 4.2:

$$\left(\frac{(B-1)^2}{U}-1\right)\cdot\frac{D}{B-2}.$$

For example, when $U = 20$GB and $B = 10$GB, the extra I/O paid amounts to the data size D, which can be quite large.

To mitigate such performance loss, we would like to acquire an accurate estimate of the key-state space size U. Since many workloads use fixed-sized states, estimating U is equivalent to estimating the number of distinct keys in the data set. In today's analytics workloads, the key space can be very large, e.g., tens of billions of URLs on the Web. So estimating the number of keys can itself consume a lot of memory. Hence, we propose to perform approximate estimation of the key space size using fast, memory-efficient "mini-analysis": state-of-the-art sketch techniques [48] can provide $1+\epsilon$ approximation for the number of distinct keys in space about $O(\epsilon^{-2})$ with low CPU overheads. Hence, given memory of modest size, these techniques can return fairly accurate approximations. Such "mini-analysis" can be applied in two ways: If the data is to be first loaded into the processing backend and later analyzed repeatedly, mini-analysis can be piggybacked in data loading with little extra overhead. For streaming workloads, such mini-analysis can be run periodically at data sources and the resulting statistics can be transferred to the MapReduce processing backend to better configure our hash algorithms. A detailed mechanism for communication between data sources and the processing backend is beyond the scope of this chapter and will be addressed in our future work.

4.1.3 A Dynamic Incremental Hash Technique (DINC-hash)

Our last technique is an extension of the incremental hash approach where we dynamically determine which keys should be processed in memory and which keys will be written to disk for subsequent processing. The basic idea behind the new technique is to recognize hot keys that appear frequently in the data set and hold their states in memory, hence providing incremental in-memory processing for these keys. The benefits of doing so are two-fold. First, prioritizing these keys leads to greater I/O efficiency since in-memory processing of data items of hot keys can greatly decrease the volume of data that needs to be first written to disks and then read back to complete the processing. Second, it is often the case that the answers for the hot keys are more important to the user than the colder keys. Then this technique offers the user the ability to terminate the processing before data is read back from disk if the coverage of data is sufficiently large for those keys in memory.

Below we assume that we do not have enough memory to hold all states of *distinct* keys. Our mechanism for recognizing and processing hot keys builds upon ideas in a widely used data stream algorithm called the FREQUENT algorithm [66, 11] that can be used to estimate the frequency of different values in a data stream. While we are not interested in the frequencies of the keys per se, we will use estimates of the frequency of each key to date to determine which keys should be processed in memory. However, note that other "sketch-based" algorithms for estimating frequencies, such as Count-Sketch [20], Count-Min [23] and CR-Precis [33], will be unsuitable for our purposes because they do not explicitly encode a set of hot keys. Rather, additional processing is required to determine frequency estimates and then use them to determine approximate hot keys, which is too costly for us to consider.

Dynamic Incremental (DINC) Hash We propose to perform dynamic incremental hash using a caching mechanism governed by the FREQUENT algorithm, i.e., to determine which tuples should be processed in memory and which should be written to disk. We use the following notation in our discussion of the algorithm: Let K be the total number of *distinct* keys. Let M be the total number of key-data item pairs in input, called tuples for simplicity. Suppose that the memory contains B pages, and each page can hold n_p key-state pairs with their associated auxiliary information. Let `up` be an update function that collapses an data item v into a state u to make a new state, $\texttt{up}(u, v)$.

Figure 4.2 illustrates the DINC-hash algorithm. While receiving tuples, each reducer divides the B pages in memory into two parts: h pages are used as write buffers, one for each of h files that will reside on disk, and $B - h$ pages for "hot" key-state pairs. Hence, the number of keys that can be processed in-memory is $s = (B - h)n_p$.[2]

The sketch of our algorithm is shown in Algorithm 1. The algorithm maintains s counters $c[1], \ldots, c[s]$, s associated keys $k[1], \ldots, k[s]$ referred to as "the keys currently being monitored", and the state $s[i]$ of a partial computation for each key $k[i]$. Initially, all the counters are set to 0, and all the keys are marked as empty (Line 1). When a new tuple (k, v) arrives, if this key is currently being monitored, the corresponding counter is incremented and the state is updated using the update function (Line 4). If k is not being monitored and $c[j] = 0$ for some j, then the key-state pair $(k[j], s[j])$ is evicted and k starts to be monitored by $c[j]$ (Line 7-8). If k is not monitored and all $c > 0$, then the tuple needs to be written to disk and all $c[i]$ are decremented by one (Line 10-11). Whenever the algorithm decides to evict a key-state pair in-memory or write out a tuple, it always first assigns the item to a hash bucket and then writes it out through the write buffer of the bucket, as in INC-hash.

[2]If we use $p > 1$ pages for each of the h write buffers (to reduce random-writes), then $s = n_p \cdot (B - hp)$. We omit p below to simplify the discussion.

Once the tuples have all arrived, most of the computation for the hot keys may have already been performed. At this point we have the option to terminate if the partial computation for hot keys is “good enough” in a sense we will make explicit shortly. If not, we proceed with performing all the remaining computation: we first write out each key-state pair currently in memory to disk to the appropriate bucket file. We then read each bucket file into memory and complete the processing for each key in the bucket file.

To compare the different hash techniques, first note that the improvement of INC-hash over the baseline MR-hash is only significant when the total number of keys K is small. Otherwise, the keys processed incrementally in main memory will only account for a small fraction of the tuples, hence limited performance benefits. DINC-hash mitigates this problem in the case when, although K may be large, some keys are considerably more frequent then other keys. By ensuring that it is these keys that are usually monitored in memory, we ensure that a large fraction of the tuples are processed before the remaining data is read back from disk.

Algorithm 1 Sketch of the **DINC-hash** algorithm

```
1: $c[i] \leftarrow 0, k[i] \leftarrow \perp$ for all $i \in \{1, 2, \cdots, s\}$
2: for each tuple $(k, v)$ from input do
3:     if $k$ is being monitored then
4:         Suppose $k$ is monitored by $c[j]$, do $c[j] \leftarrow c[j] + 1$, and $s[j] \leftarrow \texttt{update}(s[j], v)$
5:     else
6:         if $\exists j$ such that $c[j] = 0$ then
7:             Evict key-state pair $(k[j], s[j])$ to disk
8:             $c[j] \leftarrow 1$, $k[j] \leftarrow k$, and $s[j] \leftarrow v$
9:         else
10:            Write tuple $(k, v)$ to disk
11:            $c[i] \leftarrow c[i] - 1$ for all $i \in \{1, 2, \cdots, s\}$
12:        end if
13:    end if
14: end for
```

We note that besides the FREQUENT algorithm, our DINC-hash technique can also be built on a closely related variant called the Space-Saving algorithm [64]. Like the FREQUENT algorithm, Space-Saving monitors frequent items in memory and for each monitored item, maintains a counter as an estimate of the item's frequency. When the two algorithms are used as the caching mechanism in DINC-hash, they perform the same in the following two cases:

Case I: If a monitored key is received, both algorithms increment the associated counter by 1.

Case II: If an unmonitored key is received and there is at least one counter of value 0 under the FREQUENT algorithm, both algorithms evict a monitored key-state pair with the smallest counter (which must be 0 in the FREQUENT algorithm), monitor the new key-state pair with that counter, and increment the counter by 1.

The two algorithms only differ in the following case:

Case III: The arriving key is not monitored and all the counters of the monitored keys are greater than 0 in the FREQUENT algorithm. In this case, Space-Saving evicts a key-state pair with the smallest counter to make room for the new key-state pair as in Case II, whereas the FREQUENT algorithm does not take in the new key-state pair but simply decrements all counters by 1.

In most real-world workloads, the distribution of the frequencies of keys is naturally skewed. Consequently, in the FREQUENT algorithm when there is no key with counter 0, it is likely that a large number of infrequent keys have counter 1 as they appear only once in a long period of time. When Case III occurs, the counters of these infrequent keys are reduced to 0, which prevents Case III from happening again in the near future. As a result, Case III occurs infrequently under real-world workloads, and the two algorithms perform the same most of the time. We will show that the difference in performance of the two algorithms is less than 1% in the evaluation section. Due to this reason, we focus our discussion in the rest of the section on the widely used FREQUENT algorithm.

4.1.3.1 Analysis of the DINC Algorithm

We next provide a detailed analysis of the DINC algorithm.

I/O Analysis. Suppose that there are f_i tuples with key k_i. Then we have the total number of tuples $M = \sum_i f_i$. Without loss of generality assume $f_1 \geq f_2 \geq \ldots \geq f_K$. Since the memory can hold s keys, the best we can hope for is to process $\sum_{1 \leq i \leq s} f_i$ arriving tuples in memory, i.e., to collapse them into in-memory states as they are sent to the reducer. This is achieved if we know the "hot" keys, i.e., the top-s, in advance. Existing analysis for the FREQUENT algorithm can be applied to our setting to show that using the above strategy, at least M' tuples can be collapsed into states, where

$$M' = \sum_{1 \leq i \leq s} \max\left(0, f_i - \frac{M}{s+1}\right)$$

If the data is highly skewed, our theoretical analysis can be improved by appealing to a result of [11]. Specifically if the data is distributed with Zipfian parameter α our strategy guarantees that at least

$$M' = \sum_{1 \leq i \leq s} \max\left(0, f_i - \frac{M}{\max(s+1, (s/2)^{\alpha})}\right)$$

tuples have been collapsed into states. Since every tuple that is not collapsed into an existing state in memory triggers a write-out, the number of tuples written to disk is $M - s - M' + s$, where the first s in the formula corresponds to the fact that the first s keys do not trigger a write-out, and the second s comes from the write out of the hot key-state pairs in main memory. Hence, the upper bound of the number of tuples that trigger I/O is $M - M'$.

This result compares favorably with the offline optimal if there are some very popular keys, but does not give any guarantee for non-skewed data if there are no keys whose relative frequency is more than $1/(s+1)$. For example, suppose that the $r = \epsilon s$ most popular keys have total frequency $\sum_{i=1}^{r} f_i \geq (1-\epsilon)M$, i.e., a $(1-\epsilon)$ fraction of the tuples have one of r keys. Then we can conclude that

$$M' = \sum_{1 \leq i \leq s} \max\left(0, f_i - \frac{M}{s+1}\right) \geq \sum_{1 \leq i \leq r} f_i - \frac{M}{s+1} \geq (1-2\epsilon)M$$

i.e., all but a 2ϵ fraction of the tuples are collapsed into states.

Note that even if we assume the data is skewed, in INC-hash there is no guarantee on the number of tuples processed in-memory before the hash files are read back from disk. This is because the keys chosen for in-memory processing are just the first keys observed.

Sensitivity to Parameters. We further investigate whether the parameters used in the DINC-hash algorithm can affect its performance. By analyzing the memory requirements as for the INC-hash algorithm, we can obtain the following constraints:

$$\frac{U}{B-1} \leq h \leq B-1, \text{ for } B \in [\sqrt{U}+1, U]$$

where U is the total size of key-state pairs, B is the memory size, and h is the number of buckets created when evicting key-state pairs from memory to disk and writing tuples of unmonitored keys to disk. Intuitively, we again want to minimize h so that we minimize the memory consumed by the write buffers of the buckets. This way, we maximize the number of keys held in memory, s, and hence the lower bound of the number of tuples collapsed in memory, M'. The sketch-based mini-analysis proposed in the previous section can again be used here to estimate U and help set h to be its lower bound $U/(B-1)$.

However, it is worth noting that setting the number of buckets h to be its optimal value may not guarantee the optimal performance of DINC-hash. This is because DINC-hash is a dynamic caching-based scheme whose performance depends not only on the parameters used in the hash algorithm but also on the temporal locality of the keys in the input data. We detail the impact of the second factor when discussing the potential flooding behavior below.

Approximate Answers and Coverage Estimation. One of the features of DINC-hash is that a large fraction of the update operations for a very frequent key will already have been performed once all the tuples have arrived. To estimate the number of update operations performed for a given key we use the t values: these count the number of tuples that have been collapsed for key k since most recent time k started being monitored. Define the *coverage* of key k_i to be

$$\text{coverage}(k_i) = \begin{cases} t[j]/f_i & \text{if } k[j] = k_i \text{ for some } j \\ 0 & \text{otherwise} \end{cases}.$$

Hence, once the tuples have arrived, the state corresponding to k_i in main-memory represents the computation performed on a coverage(k_i) fraction of all the tuples with this key. Unfortunately we do not know the coverage of a monitored key exactly, but we know that

$$t[j] \leq f_i \leq t[j] + M/(s+1) \tag{4.3}$$

from the analysis of FREQUENT. We propose two methods to return results earlier given a user-determined threshold ϕ. For the first method, we have the following under-estimate of coverage based on Equation 4.3

$$\gamma_i := \frac{t[j]}{t[j] + M/(s+1)} \leq \frac{t[j]}{f_i} = \text{coverage}(k_i) \leq 1$$

which is very accurate when $t[j]$ or f_i is sufficiently larger that $M/(s+1)$. Hence, if $\gamma_i \geq \phi$ we can opt to return the state of the partial computation rather than to complete the computation. However, we only know if the state for key i can be output at the end of input, and we still have to spill data to disk in reducers for the keys that do not meet the coverage threshold. We can further reduce disk I/Os for the workloads that only require results of keys with frequencies higher than a threshold f^* (i.e. $f_i \geq f^*$). Based on Equation 4.3, we have the following under-estimate of coverage

$$\gamma_i' := 1 - \frac{M/(s+1)}{f^*} \leq \frac{f_i - M/(s+1)}{f_i} \leq \frac{t[j]}{f_i} = \text{coverage}(k_i) \leq 1.$$

Since γ_i' does not rely on $t[j]$, if $\gamma_i' \geq \phi$, there is no need to spill data to disk in reducers.

Potential Pathological Behaviors. Using the FREQUENT algorithm as described above leads to a deterministic policy for updating the keys being monitored. Unfortunately, by analogy to online page-replacement policies [86], it is known that any deterministic policy is susceptible to flooding and that the competitive ratio (i.e., the worst-case ratio between the number of I/O operations required by the algorithm and the optimal number) is at least a factor s, the number of keys held in memory. We note that this analysis also applies to other page-replacement policies such as LRU because they are also deterministic. For a simple example, consider the behavior of the FREQUENT algorithm on a sequence that consists of repeating the following sequence of M keys:

$$\langle k_1, k_2, \ldots, k_s, k_{s+1}, k_{s+2}, \ldots, k_{2s+1} \rangle \ .$$

The FREQUENT algorithm will require $(M - s)$ I/O operations corresponding to all but the first s terms requiring a tuple to be written to disk. In contrast, the optimal solution would be to monitor only $k_1, \ldots, k_s$ and this would require $M \cdot \frac{s+1}{2s+1}$ I/O operations.

4.1.3.2 Heuristic Improvement and the Marker Algorithm

In our implementation of FREQUENT we added a simple random heuristic that helps avoid the pathological cases described above. When evicting a currently monitored key (and its associated state), our heuristic is to randomly select from those keys whose counters are zero, rather than simply picking the first such key each time. For example, when processing

$$\langle k_1, k_2, \ldots, k_s, k_{s+1}, k_{s+2}, k_1, k_1, k_1, \ldots \rangle,$$

the heuristic means that k_1 is unlikely to be replaced by k_{s+2} (as would happen without the heuristic) due to randomization and hence we are able to combine all the k_1's in the sequence in memory. Note that the heuristic does not jeopardize the above performance guarantees. This behavior is inspired in part by the Marker algorithm [31, 63], a paging algorithm that is known to have good I/O performance (under a worst-case analysis) as a paging algorithm, but no analysis has been conducted to determine their abilities to recognize frequent keys.

The algorithm is sketched in Algorithm 2. The basic algorithm is relatively simple and can be described in terms of the difference to the FREQUENT algorithm with our heuristic: we do not increment the counters $c[i]$ beyond 1. More specifically, the algorithm maintains s bits (binary counters) $c[1], \ldots, c[s]$, s associated keys $k[1], \ldots, k[s]$ currently being monitored, and the state $s[i]$ for each key $k[i]$. Initially, all the bits are set to 0, and all the keys are marked as empty (Line 1). When a new tuple (k, v) arrives, if this key is currently being monitored, the corresponding bit is set to 1 and the state is updated using the update function (Line 4). If k is not being monitored and there exists at least one bit $c = 0$, the algorithm randomly picks a j such that $c[j] = 0$, evicts the key-state pair $(k[j], s[j])$, and let $c[j]$ monitor k (Line 7-9). If k is not monitored and all $c = 1$, then the tuple needs to be written to disk and all $c[i]$ are set to 0 (Line 11-12).

Algorithm 2 Sketch of the **Marker** algorithm

```
1:  $c[i] \leftarrow 0, k[i] \leftarrow \perp$ for all $i \in \{1, 2, \cdots, s\}$
2:  **for each** tuple $(k, v)$ from input **do**
3:      **if** $k$ is being monitored **then**
4:          Suppose $k$ is monitored by $c[j]$, do $c[j] \leftarrow 1$, and $s[j] \leftarrow \texttt{update}(s[j], v)$
5:      **else**
6:          **if** $\{i : c[i] = 0\} \neq \emptyset$ **then**
7:              $j \leftarrow$ **randomly pick** from $\{i : c[i] = 0\}$
8:              Evict key-state pair $(k[j], s[j])$ to disk
9:              $c[j] \leftarrow 1$, $k[j] \leftarrow k$, and $s[j] \leftarrow v$
10:         **else**
11:             Write tuple $(k, v)$ to disk
12:             $c[i] \leftarrow 0$ for all $i \in \{1, 2, \cdots, s\}$
13:         **end if**
14:     **end if**
15: **end for**
```

Marker versus Frequent Policies for DINC. The randomization of the Marker algorithm plays a key role in minimizing pathological behavior. Specifically it is known that competitive ratio of the enchanted Marker algorithm [63] has the competitive ratio $O(\log s)$

and that this is optimal. Note that this should not be understood to mean that the Marker algorithm is immune to flooding. However, the fact that the counters saturate at 1 in the Marker algorithm (in contrast to the unbounded counters in the FREQUENT algorithm) has a significant effect on the ability of the algorithm to identify frequent keys. In particular, if an otherwise popular key becomes temporarily infrequent, the Marker algorithm will quickly stop monitoring this key. Hence, there is no guarantee that the frequent elements are identified and therefore a policy based on the Marker algorithm would not support coverage estimation and early approximate answers as detailed above. On the other hand, the FREQUENT algorithm is more sensitive to keys have been historically popular. This is because a popular key k_i that is being monitored will have a high counter value $c[i]$ and therefore it will require the processing of at least $c[i]$ more tuples before k_i is in danger of being evicted. The extent to which it is advisable to use the historical frequency of an item to guide the monitoring of future keys is dependent on the data set. We will explore the issue further empirically in Section 4.3.3 but the summary is that because the Marker algorithm adapts more quickly to changes in the key distribution, it can end up generating more or less I/O depending on whether or not (respectively) the changes in the distribution are temporary.

4.1.4 Optimizing Other System Parameters

We finally discuss the optimization of key parameters of MapReduce systems such as the Hadoop system. We have described the optimization of Hadoop parameters under the sort-merge implementation in Section 3.3. Those key parameters, such as the chunk size C and the number of reducers R, are important to our hash-based implementation as well. In particular, when the chunk size is small, we incur high startup cost due to a large number of map tasks. An excessively large chunk size is not favorable, either. One reason is that when a combine function is used in the mappers, the hash algorithm used in the reducers is also applied in the mappers. When the chunk size is large, the output size of a map task may exceed the map buffer size and hence the hash algorithm applied to the map output incurs internal I/O spills, adding significant overheads. The second reason is that a large chunk size delays the delivery of data to the reducers, which is undesirable for incremental processing. Considering both reasons, we use the insights from Section 3.3 and set the chunk size C to the largest value that keeps the output of a map task fit in the map buffer. Regarding the number of reduce tasks R, we set it such that there is only one wave of reduce tasks, i.e. R equals the number of reduce task slots. Using multiple waves of reduce tasks incurs more I/Os because only the first wave of reduce tasks can fetch map output directly from local memory, while the reduce tasks in other waves are likely to read data from disks.

4.2 Prototype Implementation

We have built a prototype of our incremental analytics platform on Hadoop. Our prototype is based on Hadoop version 0.20.1 and modifies the internals of Hadoop by replacing key components with our Hash-based and fast in-memory processing implementations. Figure 4.4 depicts the architecture of our prototype; the shaded components and the enlarged sub-components show the various portions of Hadoop internals that we have built. Broadly these modifications can be grouped into two main components.

Hash-based Map Output Vanilla Hadoop consists of a Map Output Buffer component that manages the map output buffer, collects map output data, partitions the data for reducers, sorts the data by partition id and key (external sort if the data exceeds memory), and feeds the sorted data to the combine function if there is one or writes sorted runs to local disks otherwise. Since our design eliminates the sort phase, we replace this component with a new Hash-based Map Output component. Whenever a combine function is used, our Hash-based Map Output component builds an in-memory hash table for key-value pairs output by hashing on the corresponding keys. After the input has been processed, the values of the same key are fed to the combine function, one key at a time. In the scenario where no combine function is used, the map output must be grouped by partition id and there is

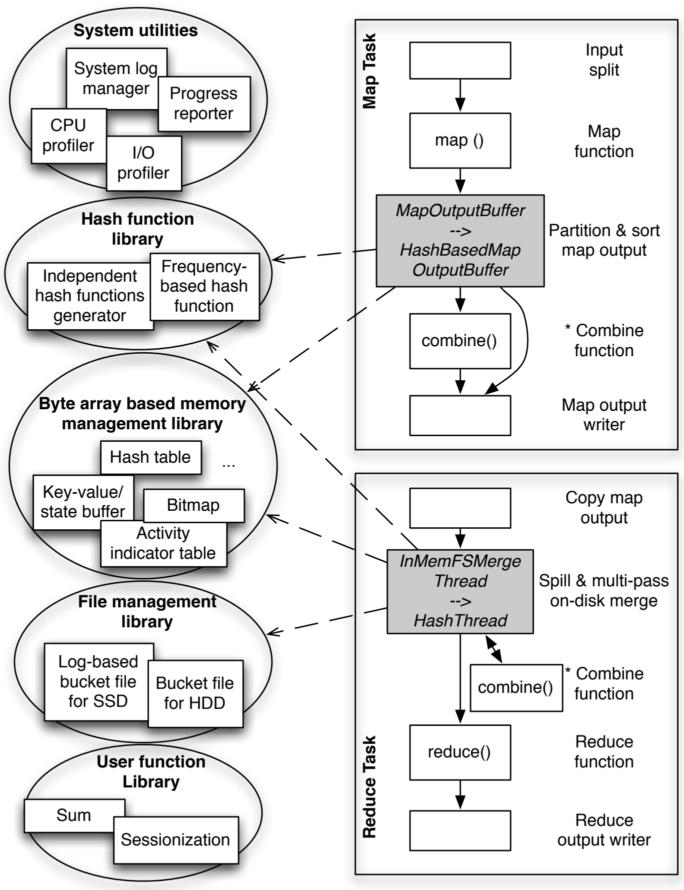


Figure 4.4. Architecture of our Hadoop-based platform for incremental processing.

no need to group by keys. In this case, our Hash-based Map Output component records the number of key-value pairs for each partition while processing the input data chunk, and moves records with the same key to a particular segment in the buffer, while scanning the buffer once.

HashThread Component. Vanilla Hadoop comprises an *InMemFSMerge* thread that performs in-memory and on-disk merges and writes data to disk whenever the shuffle buffer is full. Our prototype replaces this component with a HashThread implementation, and provides a user-configurable option to choose between MR-hash, INC-hash, and DINC-hash implementations within HashThread.

In order to avoid the performance overhead of creating a large number of Java objects, our prototype implements its own memory management by placing key data structures into byte arrays. Our current prototype includes several byte array-based memory managers to provide core functionality such as hash table, key-value or key-state buffer, bitmap, or counter-based activity indicator table, etc., to support our three hash-based approaches.

We also implement a bucket file manager that is optimized for hard disks and SSDs and provide a library of common combine and reduce functions as a convenience to the programmer. Our prototype also provides a set of independent hash functions, such as in recursive hybrid hash, in case such multiple hash functions are needed for analytics tasks. Also, if the frequency of hash keys is available a priori, our prototype can customize the hash function to balance the amount of data across buckets.

Finally, we implement several “utility” components such as a system log manager, a progress reporter for incremental computation, and CPU and I/O profilers to monitor system status.

Pipelining. In our current system, the granularity of data shuffling is determined by the chunk size (with a default value of 64MB), which is fairly small compared with typical sizes of input data (e.g., a terabyte) and hence suitable for incremental processing. In the future, if data needs to be shuffled at a higher frequency, our hash-based framework for incremental processing can be extended with the pipelining approach used in MapReduce Online [22]. The right granularity of shuffling will be determined based on the application latency requirement, the extra network overhead of fine-grained data transmission, and the existence of the combine function or not. With some of these issues addressed in MapReduce Online, we will treat them thoroughly in the hash framework in our future work.

Integration within the Hadoop Family. Our system can be integrated with other software in the Hadoop family with no or minimal effort. Any storage system in the Hadoop family, such as HBase, can serve as input to your data analytics system. This is because our internal modification to Hadoop does not require any change of input. Any query processing or data analytics system that is built over Hadoop, such as Hive, can directly benefit from our hash-based system because all our changes are encapsulated in the MapReduce processing layer. The only slight modification required in the query processing layer is transforming the reduce function for Hadoop to $init()$, $cb()$ and $fn()$ functions tailored for incremental computation. The effort of the transformation is typically minimal for analytics tasks.

4.3 Performance Evaluation

We present an experimental evaluation of our analytics platform and compare it to optimized Hadoop 0.20.1, called 1-pass SM, as described in Section 3.3. We evaluate all three hash techniques, MR-hash, INC-hash and DINC-hash, described in Section 4.1 in terms of running time, the size of reduce spill data, and the progress made in map and reduce.

In our evaluation, we use three real-world datasets: 236GB of the WorldCup click stream, 52GB of the Twitter dataset, and 156GB of the GOV2 dataset[3]. We use workloads over the WorldCup dataset: (1) **sessionization** where we split the click stream of each user into sessions; (2) **user click counting**, where we count the number of clicks made by each user; (3) **frequent user identification**, where we find users who click at least 50 times. We also use a fourth workload over both the Twitter and the GOV2 datasets, **trigram counting**, where we report word trigrams that appear more than 1000 times. Our evaluation environment is a 10-node cluster as described in Section 3.2. Each compute node is set to hold a task tracker, a data node, four map slots, and four reduce slots. In each experiment, 4 reduce tasks run on each compute node.

[3]http://ir.dcs.gla.ac.uk/test_collections/gov2-summary.htm

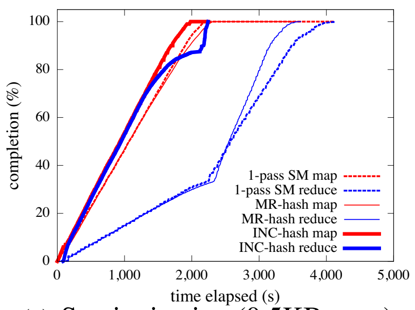
4.3.1 Small Key-state Space: MR-hash versus INC-hash

We first evaluate MR-hash and INC-hash under the workloads with small key-state space, where the distinct key-state pairs fit in memory or slightly exceed the memory size. We consider sessionization, user click counting, and frequent user identification.

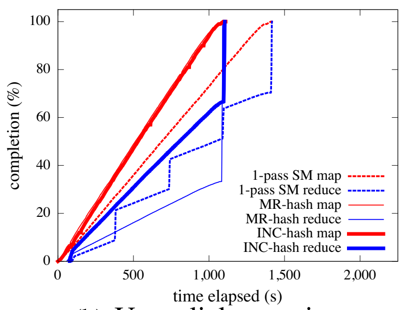
Sessionization. To support incremental computation of sessionization in reduce, we configure INC-hash to use a fixed-size buffer that holds a user's clicks. A fixed size buffer is used since the order of the map output collected by a reducer is not guaranteed, and yet online sessionization relies on the temporal order of the input sequence. When the disorder of reduce input in the system is bounded, a sufficiently large buffer can guarantee the input order to the online sessionization algorithm. In the first experiment, we set the buffer size, i.e. the state size, to 0.5KB.

Figure 4.5(a) shows the comparison of 1-pass SM, MR-hash, and INC-hash in terms of map and reduce progress. Before the map tasks finish, the reduce progress of 1-pass SM and MR-hash is blocked by 33%. MR-hash blocks since incremental computation is not supported. In 1-pass SM, the sort-merge mechanism blocks the reduce function until map tasks finish; a combine function can't be used here since all the records must be kept for output. In contrast, INC-hash's reduce progress keeps up with the map progress up to around 1,300s, because it performs incremental in-memory processing and generates pipelined output until the reduce memory is filled with states. After 1,300s, some data is spilled to disk, so the reduce progress slows down. After map tasks finish, it takes 1-pass SM and MR-hash longer to complete due to the large size of reduce spills (around 250GB as shown in Table 4.1). In contrast, INC-hash finishes earlier due to smaller reduce spills (51GB).

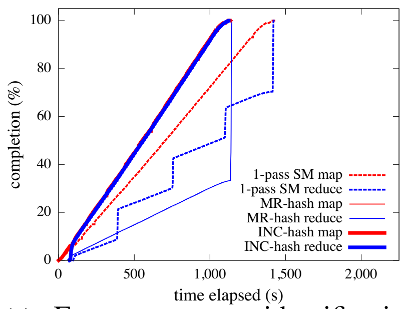
Thus by supporting incremental processing, INC-hash can provide earlier output, and generates less spill data, which further reduces the running time after the map tasks finish.



(a) Sessionization (0.5KB state).



(b) User click counting.



(c) Frequent user identification (number of clicks ≥ 50).

Figure 4.5. Progress report using hash implementations.

User click counting & Frequent user identification. In contrast to sessionization, user-click counting can employ a combine function and the states completely fit in memory at the reducers.

Figure 4.5(b) shows the results for user click counting. 1-pass SM applies the combine function in each reducer whenever its buffer fills up, so its progress is more of a step function. Since MR-hash does not support the combine function, its overall progress only reaches 33% when the map tasks finish. In contrast, INC-hash makes steady progress through 66% due to its full incremental computation. Note that since this query does not allow any early output, no technique can progress beyond 66% until all map tasks finish.

This workload generates less shuffled data, reduce spill data, and output data when compared to sessionization (see Table 4.1). Hence the workload is not as disk- and network-I/O- intensive. Consequently both hash-based techniques have shorter running times, when compared to 1-pass SM, due to the reduction in CPU overhead gained by eliminating the sort phase.

We further evaluate MR-hash and INC-hash with frequent user identification. This query is based on user click counting, but allows a user to be output whenever the counter of the user reaches 50. Figure 4.5(c) shows 1-pass SM and MR-hash perform similarly as in user click counting, as the reduce function cannot be applied until map tasks finish. The reduce progress of INC-hash completely keeps up with the map progress due to the ability to output early.

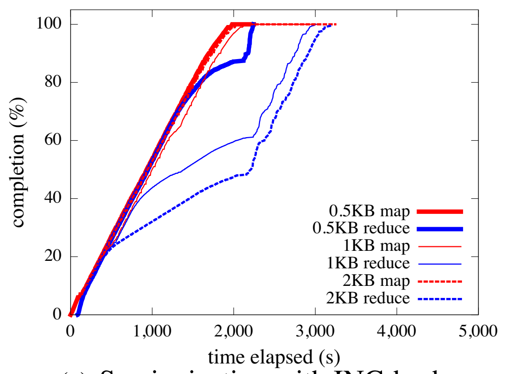
In summary, given sufficient memory, INC-hash performs fully in-memory incremental processing, due to which, its reduce progress can potentially keep up with the map progress for queries that allow early output. Hash techniques can run faster if I/O and network are not bottlenecks due to the elimination of sorting.

4.3.2 Large Key-state Space: INC-hash versus DINC-hash

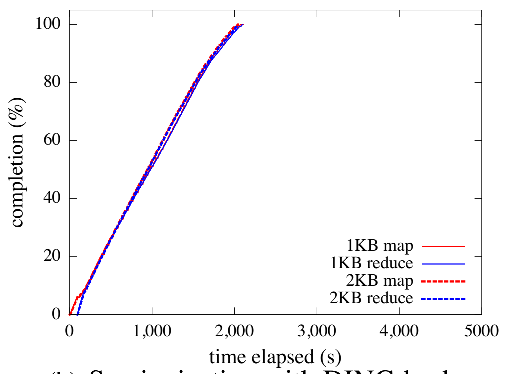
We next evaluate INC-hash and DINC-hash for incremental processing for workloads with a large key-state space, which can trigger substantial I/O. Our evaluation uses two workloads below:

Sessionization with varying state sizes. Figure 4.6(a) shows the map and reduce progress of INC-hash under three state sizes: 0.5KB, 1KB, and 2KB. A larger state size means that the reduce memory can hold fewer states and that the reduce progress diverges earlier from the map progress. Also, larger states cause more data to be spilled to disk, as shown in Table 4.2. So after map tasks finish, the time for processing data from disk is longer.

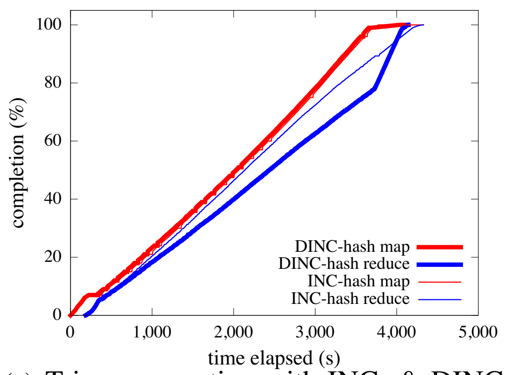
To enable DINC-hash for sessionization, a streaming workload, we use the state of a monitored key to hold the clicks of a user in her recent sessions. We evict a state from memory if: (1) all the clicks in the state belong to an expired session; (2) the counter of the state is zero. Rather than spilling the evicted state to disk, the clicks in it can be directly output. As shown in Table 4.2, DINC-hash only spills 0.1 GB data in reduce with 2KB state



(a) Sessionization with INC-hash.



(b) Sessionization with DINC-hash.



(c) Trigram counting with INC- & DINC-hash.

Figure 4.6. Progress report using hash implementations.

size, in contrast to 203 GB for the same workload in INC-hash. As shown in Figure 4.6(b), the reduce progress of DINC-hash closely follows the map progress, and spends little time processing the on-disk data after mappers finish.

We further quote numbers about stock Hadoop for this workload (see Table 3.1). Using DINC-hash, the reducers output continuously and finish as soon as all mappers finish reading the data in 34.5 minutes, with 0.1GB internal spill. In contrast, the original Hadoop system returns all the results towards the end of the 76 minute job, causing 370GB internal data spill to disk, 3 orders of magnitude more than DINC-hash.

Trigram Counting. Figure 4.6(c) shows the map and reduce progress plot for INC-hash and DINC-hash with the Gov2 dataset. The reduce progress in both keeps growing below, but close to the map progress, with DINC-hash finishing a bit faster. In this workload, the

Table 4.1. Comparing optimized Hadoop (using sort-merge), MR-hash, and INC-hash.

Sessionization	1-Pass SM	MR-hash	INC-hash
Running time (s)	4135	3618	2258
Map CPU time per node (s)	1012	659	571
Reduce CPU time per node (s)	518	566	565
Map output / Shuffle (GB)	245	245	245
Reduce spill (GB)	250	256	51
User click counting	1-Pass SM	MR-hash	INC-hash
Running time (s)	1430	1100	1113
Map CPU time per node (s)	853	444	443
Reduce CPU time per node (s)	39	41	35
Map output / Shuffle (GB)	2.5	2.5	2.5
Reduce spill (GB)	1.1	0	0
Frequent user identification	1-Pass SM	MR-hash	INC-hash
Running time (s)	1435	1153	1135
Map CPU time per node (s)	855	442	441
Reduce CPU time per node (s)	38	38	34
Map output / Shuffle (GB)	2.5	2.5	2.5
Reduce spill (GB)	1.1	0	0

Table 4.2. Comparing sessionization to INC-hash with 0.5KB state, INC-hash with 2KB state, and DINC-hash with 2KB state.

	INC (0.5KB)	INC (2KB)	DINC (2KB)
Running time (s)	2258	3271	2067
Reduce spill (GB)	51	203	0.1

reduce memory can only hold 1/30 of the states, but less than half of the input data is spilled to disk in both approaches. This implies that both hash techniques hold a large portion of hot keys in memory. DINC-hash does not outperform INC-hash like with sessionization because the trigrams are distributed more evenly than the user ids, so most hot trigrams appear before the reduce memory fills up. INC-hash naturally holds them in memory. The reduce progress in DINC-hash falls slightly behind that of INC-hash because if the state of a key is evicted, and the key later gets into memory again, the counter in its state starts from zero again, making it harder for a key to reach the threshold of 1,000. Both hash techniques finish the job in the range of 4,100-4,400 seconds. In contrast, 1-pass SM takes 9,023 seconds. So both hash techniques outperform Hadoop.

In summary, for workloads that require a large key-state space, our frequent-key mechanism significantly reduces I/Os and enables the reduce progress to keep up with the map progress, thereby realizing incremental processing

4.3.3 Dynamic Hashing with Alternative Caching Algorithms

We have compared INC-hash with dynamic hashing based on the deterministic FREQUENT algorithm. Now we further extend the evaluation to dynamic hashing technique based on the randomized Marker algorithm, as described in Section 4.1.3. We refer the two dynamic hashing techniques as DINC-Frequent, and DINC-Marker, respectively. We compare the sizes of reduce intermediate files across these schemes. We consider three workloads: trigram counting on the Twitter dataset, trigram counting on the Gov2 dataset, and sessionization on the WorldCup dataset with 2KB state.

Different Workloads. To enable an in-depth analysis of these hashing techniques, we characterize our workloads as in one of the following two types:

- ► *Non-streaming*: When a key-state pair is evicted from memory, it is written to a reduce intermediate file for further processing. Trigram counting belongs to this type of workload. If a trigram is evicted, its state, which is a counter, is written to disk and later summed with other counters for the same trigram.
- ► *Streaming*: For streaming workloads, the decision to monitor a new key in memory is based both on the existence of a state containing complete data for reduce processing, e.g., when a window closes, and on how the paging algorithm used in each hashing technique finds such a state in memory to evict: (1) For DINC-Frequent, the states with zero counters are scanned to find a qualified state with complete data for reduce processing. (2) For DINC-Marker, the states with zero markers are scanned in a random order to find a qualified state. (3) INC-hash does not allow any state to be replaced. If such a key-state pair exists for eviction, it is output directly but not written to a file on disk. Sessionization belongs to this type of workload. For each user, its

Table 4.3. Comparing INC-hash, DINC-Frequent and DINC-Marker in terms of reduce spill (GB).

Workload	INC-hash	DINC-Frequent	DINC-Marker
Trigram counting (Twitter)	318	288	316
Trigram counting (Gov2)	221	176	188
Sessionization (2KB)	203	0.1	0

state is a window containing the user's recent clicks. The window closes when the user's click session is considered complete. If a user's state needs to be replaced, it is directly output at the time of eviction.

As shown in Table 4.3, for the non-streaming workloads, i.e. trigram counting using Twitter and Gov2 datasets, DINC-Frequent has the smallest intermediate file size because it maintains more history information and is thus more effective to identify hot key than the others. For sessionization, the streaming workload, the two dynamic hashing techniques perform remarkably better than INC-hash. INC-hash incurs significant intermediate I/O because it does not allow a state to be replaced, and thus all the input tuples with keys not in memory are written to files. Both dynamic hashing techniques have negligible intermediate I/O. Their I/O numbers are slightly different due to the way we use them to find a qualified state to evict. It does not show intrinsic difference of the two techniques.

Effects of Locality and Memory Size. We further evaluate our hashing techniques under different data locality properties and different memory sizes using trigram counting on the Twitter dataset. To explore different localities, we construct two datasets: (1) In the original dataset, tweets in various languages are mixed together. (2) The manipulated dataset is generated from the original dataset. We group the tweets that belong to 8 language families into 8 subsets. The set of trigrams from one subset hardly overlaps that from another subset. The subsets are arranged sequentially in the input to each algorithm, and thus show strong locality. This type of locality is similar to the evolving trends of hot topics in real world social networks. In our evaluation we also explore the effect of the memory size by using two settings: *Constrained memory* with 100MB per reducer, which can hold 2% of all key-states across all reducers; *Larger memory* with 600MB per reducer, which can hold 11%-15% of all key-states, depending on the meta data size in different techniques.

Besides DINC-Frequent and DINC-Marker, we also take this opportunity to report the performance of dynamic hashing using another deterministic algorithm, the Space-Saving algorithm (referred to as DINC-SS), as we vary the data locality and memory size. For fair comparison, each technique allocates the given memory for both key-state pairs and meta data. Meta data includes the data structures that organize key-states as a hash-table in order to efficiently search any state by its key, and additional data structures for realizing a replacement policy efficiently. All the techniques share the same meta data for the hash table. The sizes of additional meta data for cache management vary. INC-hash has no additional meta data since it does not support state replacement. DINC-Frequent maintains a set of counters with distinct values, and the two-way mapping between a key-state and its corresponding counter. The implementation of DINC-SS does not include the data structures in the Space-Saving algorithm that do not affect the replacement policy. As a result, DINC-SS has the same data structures for meta data as DINC-Frequent. DINC-Marker maintains the two-way mapping between a key-state and two static markers: 0 and 1. So, the comparison of meta data sizes of the three techniques is: INC-hash $<$ DINC-Marker $<$ DINC-Frequent $=$ DINC-SS.

The comparison of intermediate file sizes is shown in Table 4.4. In all the combinations of data locality types and memory sizes, DINC-Frequent and DINC-SS differ less than 1% in terms of intermediate I/Os. The reason is the intrinsic similarity of the two algorithms as explained in Section 4.1.3. With constrained memory size, DINC-Frequent/SS outperforms the others on both the original dataset and the manipulated dataset. This is because DINC-Frequent/SS recognizes hot keys more efficiently based on the history memorized by the counters, and thus makes better use of limited memory. It is also noticeable that the advantage of DINC-Frequent/SS over DINC-Marker is less with the manipulated dataset. This shows the trade-off between DINC-Frequent/SS and DINC-Marker. On one hand,

Table 4.4. Comparing INC-hash, DINC-Frequent, DINC-SS and DINC-Marker with the trigram counting workload over the Twitter dataset in terms of reduce spill (GB).

Language	Memory per reducer(MB)	INC-hash	DINC-Frequent (DINC-SS)	DINC-Marker
Mixed	100	318	**288** (**288**)	316
Separate	100	502	**266** (**265**)	272
Mixed	600	**217**	241 (241)	244
Separate	600	455	251 (251)	**212**

DINC-Frequent/SS is still better at identifying hot keys than DINC-Marker. On the other hand, as stated in Section 4.1.3, when a subsequent subset of data is read from the manipulated dataset, the states from the previous subset are protected by counters in memory. Since the keys from the new subset are different from the old subset, some tuples from the new subset are written to disk due to this reason until the counters of old states become zero. To the contrary, DINC-Marker protects states with markers, which are boolean counters. So, DINC-Marker is more I/O efficient in the process of clearing states from an old subset for a new subset. We will show this trade-off again with the larger memory setting.

In the larger memory setting using the original dataset, INC-hash performs the best. INC-hash covers the hot keys very well for two reasons. First, there is no strong locality in the dataset. That is, the distinct keys are quite evenly distributed in the dataset. Second, the memory is sufficiently large such that most hot keys are likely to show up in the first s keys. In this case, since INC-hash can hold more keys than the others due to the smallest meta data size as explained before. Hence, it has the least intermediate I/O. In the larger memory setting using the manipulated dataset, INC-hash becomes much worse and DINC-Marker performs the best. INC-hash incurs more I/O because the first s keys do not cover hot keys well due to data locality. DINC-Marker outperforms DINC-Frequent/SS, demonstrating their trade-off again. But with larger memory, the advantage of DINC-Frequent/SS on identifying hot keys is not as significant as before. Thus, DINC-Marker outperforms DINC-Frequent/SS.

In this set of experiments, we show that for streaming workloads, all dynamic hashing techniques have significantly less intermediate I/O than INC-hash. For non-streaming

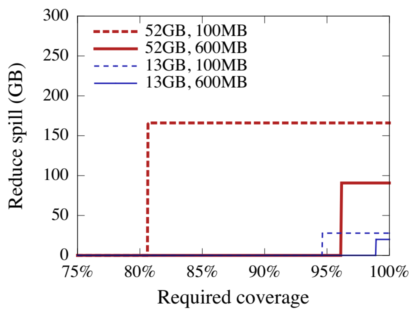


Figure 4.7. Disk spills of reducers in Twitter trigram counting when we tune the coverage requirement for the trigrams that occur more than 1000 times, under different input data sizes and reducer memory sizes.

workloads with constrained memory, DINC-Frequent (or similarly, DINC-SS) outperforms the others because it recognizes hot keys more efficiently and thus makes better use of limited memory. For non-streaming workloads with larger memory, when keys are evenly distributed in the data INC-hash outperforms the others due to the early appearance of hot keys and the small meta data used, whereas in the presence of strong locality in the key distribution, dynamic hashing works better than INC-hash and in particular DINC-Marker performs the best for the type of locality of evolving trends.

4.3.4 Dynamic Hashing under Coverage Requirements

We now evaluate the DINC-hash optimization technique in Section 4.1.3.1 for workloads with coverage requirements. Figure 4.7 shows the disk spills in reducers in the Twitter trigram counting workload when we tune the coverage requirement for the frequent trigrams (occur more than 1000 times) under different input data sizes (13GB and 52GB) and reducer memory sizes (100MB and 600MB per reducer). Under 52GB input data and 100MB memory size, up to 80% coverage can be supported without reduce spill. In all the other settings, at least 94% coverage can be supported without incurring any reduce spill.

4.3.5 Validation using Amazon EC2

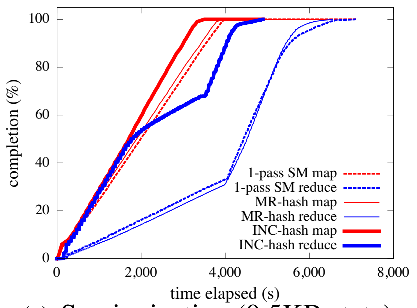
We further validate our evaluation results using much larger datasets in the Amazon Elastic Compute Cloud (EC2). Our EC2 cluster consists of 20 standard on-demand extra large instances (4 virtual cores and 15GB memory per instance) launched with 64-bit Amazon Linux AMI 2012.03 (based on Linux 3.2). One of the instances serves as a head node running the name node and the job tracker. The other 19 instances serve as slave nodes, where each node carries a data node and a task tracker. Each slave node is attached with three Elastic Block Store (EBS) volumes: an 8GB volume for the operating system, a 200GB volume for HDFS, and another 200GB volume for intermediate files. Each slave node is configured with 3 map task slots and 4 reduce task slots.[4] Due to the constraints of the physical memory size and the EC2 setting, the maximum buffer size that we can set for a map task or a reduce task is slightly more than 700MB of memory.[5] Therefore, we set the buffer size for a map or reduce task to be 700MB by default, unless we intentionally reduce the buffer size to evaluate our techniques in the case of constrained memory.

Click Stream Analysis We first validate the results of click stream analysis on EC2. We use 1TB of the WorldCup click stream, the size of which is increased by a factor of 4 in order to keep the data-to-memory ratio close to that in our previous experiments.

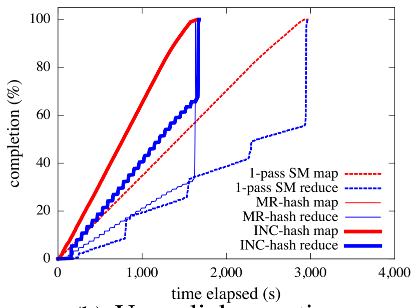
The first set of experiments compare INC-hash with 1-pass SM and MR-hash in the case that memory is sufficiently large for holding most (but not all) key-state pairs. The progress plots of three workloads, namely sessionization, user click counting, and frequent user identification, are shown in Figure 4.8(a), 4.8(b) and 4.8(c), respectively. These results

[4]Due to the limited performance of a virtual core in EC2 (in contrast to a real core in our previous cluster), the data shuffling progress cannot keep up with the map progress under the setting of 4 map task slots and 4 reduce task slots per node, which was used in our previous experiments. To solve the problem, we set 3 map task slots and 4 reduce task slots per node in EC2.

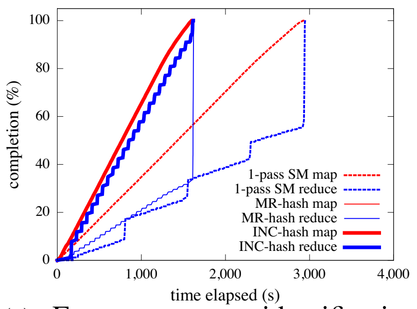
[5]The details of our memory allocation scheme are the following: On a slave node, we are able to allocate 1.4GB JVM heap space for each of the 7 map and reduce task slots, while reserve the remaining memory for the data node, task tracker and the other services in the operating system. And we are able to allocate about half of the JVM heap space, 700MB per task, as the buffer.



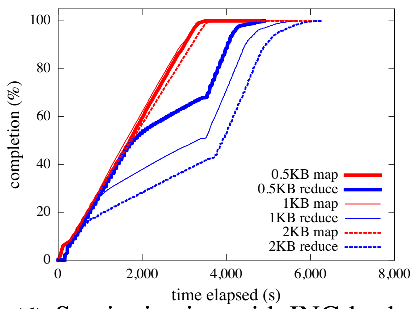
(a) Sessionization (0.5KB state).



(b) User click counting.



(c) Frequent user identification (number of clicks ≥ 50).



(d) Sessionization with INC-hash.

Figure 4.8. Progress report using hash implementations on EC2.

agree with the observations in Section 4.3.1: (1) When the memory is sufficiently large, INC-hash is able to perform incremental processing fully in memory, which allows the reduce progress to get closer to the map progress when early output can be generated for the query; this is shown in Figure 4.8(a) for sessionization before the map progress reaches 50% and in Figure 4.8(c) for frequent user identification over the entire job. (2) The INC-hash technique runs faster than 1-pass SM and MR-hash due to the elimination of sorting and less I/O, which can be seen in all of the three experiments.

We next compare the performance of INC-hash (Figure 4.8(d)) and that of DINC-hash (Figure 4.9) using the sessionization workload that involves a large key-state space far exceeding available memory. These results validate our observations in Section 4.3.2: When the memory cannot hold all key-state pairs, DINC-hash dramatically reduces I/O to realize incremental processing, and enables the reduce progress to keep up with the map progress.

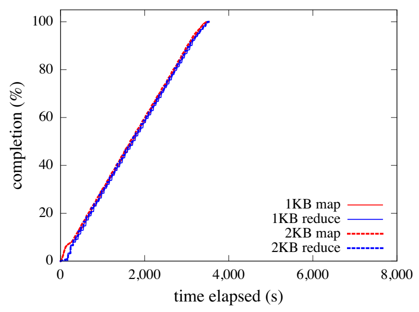


Figure 4.9. Sessionization with DINC-hash on EC2.

The results on EC2 also show some different characteristics. We list the performance measurements of the above experiments in Table 4.5 and Table 4.6. (1) *Higher CPU cost:* All the experiments have higher per-node CPU cost than our previous experiments, as shown in Table 4.5. This is because each node on EC2 processes more data whereas the virtual cores on each node have less processing power than the real cores used in our previous experiments. (2) *More I/O for the hash-based techniques:* Comparing the reduce spill numbers in Table 4.6 and the corresponding numbers in Table 4.2, we can see that the I/O cost of reduce spill on EC2 increases by a factor ranging between 4.8 and 10.7, more than the factor 4 by which the data increases. This is because the total buffer size on EC2 is less than 4 times the buffer size in our previous experiments, as we explained earlier, which aggregates the memory pressure and causes more I/Os. (3) *Longer running time:* As shown in Table 4.5 and Table 4.6, the running times of all the experiments are longer than our pervious experiments. Both the increase in CPU cost and the increase in I/O contribute to the longer running times. We finally note that we also encountered unexpected CPU measurements largely due to the interference from other jobs, such as the reduce CPU time of MR-hash for sessionization in Table 4.5, which illustrates the difficulty of doing CPU studies in shared environments like EC2.

Trigram Counting We next validate the results of trigram counting using the Twitter dataset. The goal is to evaluate different hash-based techniques under different data locality

Table 4.5. Comparing optimized Hadoop (using sort-merge), MR-hash, and INC-hash on EC2.

Sessionization	1-Pass SM	MR-hash	INC-hash
Running time (s)	6605	6694	4895
Map CPU time per node (s)	1872	1121	1112
Reduce CPU time per node (s)	1353	3080	1648
Map output / Shuffle (GB)	1087	1087	1087
Reduce spill (GB)	1048	1126	547
User click counting	1-Pass SM	MR-hash	INC-hash
Running time (s)	2965	1641	1674
Map CPU time per node (s)	1949	960	960
Reduce CPU time per node (s)	107	91	84
Map output / Shuffle (GB)	8.5	8.5	8.5
Reduce spill (GB)	4.8	0	0
Frequent user identification	1-Pass SM	MR-hash	INC-hash
Running time (s)	2951	1626	1616
Map CPU time per node (s)	1953	961	958
Reduce CPU time per node (s)	93	82	84
Map output / Shuffle (GB)	8.5	8.5	8.5
Reduce spill (GB)	4.8	0	0

Table 4.6. Comparing sessionization to INC-hash with 0.5KB state, INC-hash with 2KB state, and DINC-hash with 2KB state on EC2.

	INC (0.5KB)	INC (2KB)	DINC (2KB)
Running time (s)	4895	6252	3523
Reduce spill (GB)	547	978	0.9

Table 4.7. Comparing INC-hash, DINC-Frequent and DINC-Marker with the trigram counting workload over the Twitter dataset on EC2 in terms of reduce spill (GB).

Language	Memory per reducer(MB)	INC-hash	DINC-Frequent	DINC-Marker
Mixed	100	825	**754**	820
Separate	100	1189	**597**	612
Mixed	700	**557**	581	572
Separate	700	1074	542	**428**

properties and reduce buffer sizes. Since the maximum total size of reduce buffers on EC2 is approximately 2.25 times that in our previous experiments, in order to validate results using the same data-to-buffer ratio, we also increase the input data by a factor of 2.25, to 117GB. Similar to our previous experiments, we use two datasets: the *Original dataset*, where tweets in various languages are mixed together; and the *Manipulated dataset*, where the tweets are grouped into multiple language families and fed into each algorithm sequentially. We also vary the reduce buffer size using two settings: *Constrained memory* with 100MB per reducer, and *Larger memory* with 700MB per reducer. The comparison of intermediate file sizes is shown in Table 4.7. In the constrained memory setting, DINC-Frequent performs the best on both datasets. In the larger memory setting using the original dataset, INC-hash outperforms the other two. In the larger memory setting using the manipulated dataset, DINC-Marker performs the best. The results agree with our previous experiments, as summarized in Section 4.3.3.

Summary. The results in this section are summarized below:

► Our incremental hash technique provides much better performance than optimized Hadoop using sort-merge and the baseline MR-hash: INC-hash significantly improves the progress of the map tasks, due to the elimination of sorting, and given sufficient memory, enables fast in-memory processing of the reduce function.

► For a large key-state space, dynamic hashing based on frequency analysis can significantly reduce intermediate I/O and enable the reduce progress to keep up with the map progress for incremental processing.

- ► For streaming workloads, our dynamic hashing techniques can provide up to 3 orders of magnitude reduction of intermediate I/O compared to other techniques.
- ► For non-streaming workloads, there are trade-offs between hashing techniques depending on the data locality and memory size. Dynamic hashing using the frequency analysis tends to work well under constrained memory. When there is sufficiently large memory, other hashing techniques may perform better for various data locality properties.

4.4 Related Work

The FREQUENT algorithm [66, 11], also known as the Misra-and-Gries algorithm, is a deterministic algorithm to identify ϵ-approximate frequent items in a entire data stream with $1/\epsilon$ counters. Each counter corresponds to an item monitored in memory, and gives an underestimate about the frequency of the item. The algorithm makes no assumption on the distribution of the item frequencies, and can be implemented such that each update caused by a data item takes $O(1)$ time. The Space-Saving algorithm [64] employs a mechanism similar to that in the FREQUENT algorithm. Each counter in Space-Saving gives an overestimate about the frequency of the corresponding monitored item. In addition, an upper bound for the error of each counter from the true frequency is maintained. Thus, Space-Saving is also able to give an underestimate of the frequency for each monitored item. However, when Space-Saving is used as a paging algorithm, the paging behavior relies only on the counters, but not affected by the error bounds.We explained the similar performance for paging between the above algorithms in Section 4.1.3 and exhibited it empirically in Section 4.3.3.

The algorithm proposed in [56] aims to find frequent elements in sliding windows, which combines the FREQUENT algorithm with another existing algorithm. Nevertheless, the update time scales with the number of monitored keys, and hence renders poor performance without window operators. This work is related to our future work of extending our incremental analytics platform to support stream processing.

4.5 Summary

In this chapter, we proposed a new MapReduce platform that employs a purely hash-based framework, with various techniques to enable incremental processing and fast in-memory processing for frequent keys. Evaluation of our Hadoop-based prototype showed that it can significantly improve the progress of map tasks, allows the reduce progress to keep up with the map progress with up to 3 orders of magnitude reduction of internal data spills, and enables results to be returned early.

CHAPTER 5

STREAM PROCESSING WITH LATENCY CONSTRAINTS

We have removed the intrinsic blocking operator in the original MapReduce implementation. The revised MapReduce platform offers data parallelism and incremental processing. We ask the following two questions. Are data parallelism and incremental processing enough for streaming analytical queries with stringent latency requirements? If not, which additional design features are needed? As discussed in Chapter 1, many existing systems tailored for "fast data" also provide data parallelism, and target at incremental processing in ways different from ours: they either impose the significant overhead of implementing incremental processing to users, or only support incremental processing efficiently when memory is sufficient. These systems share the same questions as we asked here.

In order to answer the first question, we understand the sources of latency by conducting a benchmark study over our revised Hadoop platform (Incremental Hadoop). The study reveals that while incremental processing allows arriving tuples to be processed one-at-a-time, it does not guarantee the actual latency of processing each tuple in a large distributed system for two reasons. (1) Upstream and downstream operators may process data at different speeds, causing high latency due to substantial data accumulation in between. Hence, it is crucial to consider workload characteristics of each job, and allocate resource to operators by configuring the *degree of parallelism* (e.g., the number of processes per node) and *granularity of scheduling* (e.g., batching data items every 5ms for shuffling) in each operator. (2) When the memory of a cluster is not large enough to process all data in memory, the tuples spilled to disk experience high latency because their processing is often deferred to a later phase (e.g., at the end) of the job. These two key observations call for *job-specific resource planning* to select the appropriate parameter settings and *latency-aware scheduling* to determine which tuples to process and in what order to process them in order to keep latency low. In this thesis, we propose resource planning and runtime scheduling techniques as summarized below.

Model-driven Resource Planning. We propose a model-driven approach to automatically determining the resource allocation plan for each job. We formulate the per-job resource planning problem as a constrained optimization problem: given a user analytic job and latency constraint $\mathcal{L}$, find a resource allocation plan to maximize throughput while subjecting the latency of results to $\mathcal{L}$. We model a variety of latency metrics, including per-tuple latency, per-window latency, and any quantiles associated with these latency distributions. Then any of these latency metrics can be subjected to the user latency constraint. To develop this collection of models, we identify and analyze the dominant components of latency in a complex distributed system. This involves two challenges: First, latency here covers a wide range of data processing behaviors, e.g., for processing individual tuples, producing windowed results from a set of tuples, and capturing the variation of each type of latency. Second, a distributed system based on data parallelism also exhibits complex system behaviors. Our major contribution here lies in a clean extraction of dominant data processing and system-level behaviors from the Incremental Hadoop with a maze of complexity.

Latency-aware Scheduling. We further propose latency-aware scheduling at runtime to determine the set of tuples to process and the order to process them in order to maximize the number of results that meet the latency requirement, i.e., the *total utility*. Such runtime scheduling is helpful because at runtime, the workload characteristics may differ from those provided earlier to my model-driven resource planning, e.g., due to bursty inputs and change of computation costs under constrained memory. Thus, runtime selection and prioritization of tuples greatly affects the overall utility. We propose two runtime scheduling algorithms, at batch-level and tuple-level, respectively, which consider both costs and deadlines of data processing. In particular, our tuple-level scheduling algorithm has provable results on the quality of runtime schedules and efficiency of the scheduling algorithm.

Evaluation using real-world workloads such as click stream analysis and tweet analysis show the following results: (1) Our models can capture the trend of actual latency changes when tuning system parameters, with error rates within 15% for the average latency metric, and within 20% for 0.9- and 0.99-quantiles of latency. (2) Our model-driven approach to resource planning can reduce the average latency from 10's of seconds in Incremental Hadoop to sub-second, with 2x-5x increase in throughput. (3) For runtime scheduling, our latency-aware tuple scheduling algorithm outperforms D^{over} [52], a state-of-the-art scheduling algorithm with provable optimality in the worse case, and can dramatically improve the number of tuples meeting the latency constraint, especially under constrained memory. (4) We finally compare our system to **Twitter Storm** [92] and **Spark Streaming** [99], two state-of-the-art distributed stream systems. For all workloads tested, our system offers *1-2 orders of magnitude* improvements over Storm and Sparking Streaming when considering both latency and throughput.

This chapter is organized as follows. We first show our benchmark study in Section 5.1 and some necessary architectural changes in Section 5.2. We then introduce our model-based resource planning in Section 5.3 and runtime scheduling in Section 5.4. Finally, we present the evaluation results in Section 5.5, and conclude in Section 5.7.

5.1 An Initial Benchmark Study

We begin by benchmarking Hadoop and our Hadoop-based incremental processing platform to understand the causes of latency. The key observations from this study motivated us to propose necessary architectural changes and new techniques in later sections which dramatically boost performance.

In this study, we consider analytical queries in the domains of (a) Twitter feed analysis, for which we collected 13GB of tweets; (b) click stream analysis, for which we collected

Table 5.1. Streaming workloads in Twitter feed and click analysis.

Type of Workload		*Twitter Feed*[55, 65]	*Click Stream*[65]
Incre-mental Update	**Aggr** (1)	– word frequency – bi/tri-gram counting – co-occurrence of words	– url frequency – per-user counting – per-topic counting
	UDF (2)	– reputation score for each user as he tweets	– click stream sessionization
Windowed Operations	**Aggr** (3)	windowed version of workload type (1)	
	UDF (4)	windowed version of workload type (2)	

236GB of WorldCup click data (described in Section 3.2). We identified streaming workloads from recent studies in these domains [55, 65], which are based on real-world applications. We classify these workloads in four types, as summarized in Table 5.1. For now, we focus on the first two types under the category of "incremental update" (while postponing the discussion of windows to the next section). These workloads follow a general pattern: map() extracts key-value pairs (tuples) and performs filtering; the tuples are then grouped by the key; finally reduce() performs analytics within each group. Type-1 workloads use standard aggregates in reduce(), such as counting the frequency of each word or each tri-gram in tweets. "Incremental update" means that the state for a key (e.g., a counter) is updated incrementally when a new tuple arrives. The state of a key can be output based on a user-specified frequency or when the counter exceeds a threshold. Type-2 workloads differ by using a user-defined function (UDF) in reduce(). Examples include computing the co-occurrence of each pair of words in user-defined contexts [65], or breaking clicks into sessions based on user-defined criteria. The state of a key (e.g., the current session of a user) can still be updated when a tuple arrives.

To measure latency, we break down the lifetime of a *tuple*, a key-value pair output by the map function, into a number of phases.

- ▶ Map Function (MF): This phase begins when the input data chunk that contains the tuple is read, and completes when the chunk is fully processed by map().
- ▶ Map Materialization (MM): This phase begins after MF and completes when the map output is completely written to disk.

- ▶ Shuffle Wait (SW): This phase starts after MM and ends when the map output containing the tuple starts to be shuffled.
- ▶ Shuffle Transfer (ST): This phase starts after SW and captures the actual time spent during shuffling.
- ▶ Reduce Wait (RW): This phase is after ST and before the tuple is consumed by the reduce function.
- ▶ Reduce Update (RU): The cost of updating a state in-memory using this tuple in reduce().

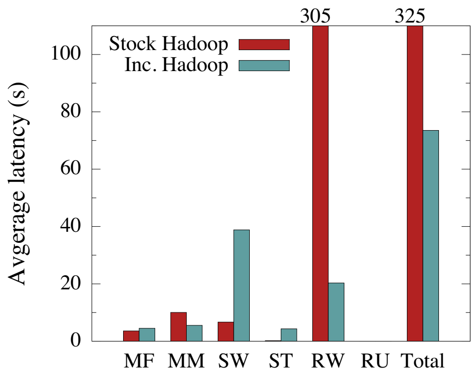
We sum the time measurements of these phases as the *tuple latency*.

We perform the benchmark study using a cluster of 10 compute nodes, each equipped with an Intel Xeon X3360 Processor (4 cores), 8GB RAM, and a 2TB Western Digital RE4 HDD. We run the DataNode and TaskTracker daemons on each compute node, and configure 3 map tasks and 3 reduce tasks running in parallel per node. In each job, only one wave of reduce tasks is used (30 reduce tasks).[1] The HDFS block size is 32MB. We compare stock Hadoop (sort-merge) and Incremental Hadoop (INC-hash that we proposed in Chapter 4). Both approaches are based on Hadoop 0.20.1 for fair comparison.

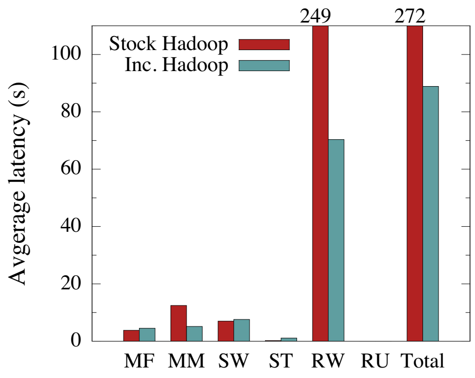
We first consider the case that the aggregate memory in reducers can hold all key-state pairs in memory. We illustrate our main observations using a word counting query (type-1 workload) over the Twitter dataset. Figure 5.1(a) shows the breakdown of average tuple latency of the two systems. Compared to Incremental Hadoop, stock Hadoop cannot apply the reduce function until all mappers complete, causing high delay in the Reduce Wait phase and hence not suitable for low-latency analytics as reported in Chapter 3. Now we focus on Incremental Hadoop and look into its latency issues.

(1) *Latency Caused by Data Accumulation.* Incremental Hadoop has significant latency in both Shuffle Wait (SW), e.g., 39 sec, and Reduce Wait (RW), e.g., 20 sec. The reason

[1]We manually tuned the configuration to achieve best performance for these workloads. The effect of configuration on latency is studied in detail in Section 5.3.



(a) Latency breakdown of stock Hadoop and Incremental Hadoop w/o reduce spill.



(b) Latency breakdown of stock Hadoop and Incremental Hadoop with reduce spill.

Figure 5.1. A latency benchmark study of stock Hadoop and Incremental Hadoop.

is that reducers cannot keep up with mappers in processing, and hence data accumulates between them. Specifically, as map output fills up a large shuffle buffer, it takes long for the reducer to process the data, causing increased RW latency. Further, when the shuffle buffer becomes full, the reducer will not pull more data from newly completed mappers — such blocking of shuffling from mappers causes high SW latency.

(2) *Latency Caused by a Large Chunk Size.* For Incremental Hadoop, each phase of a tuple's lifetime contains at least seconds of latency. This is caused by the relative large input chunk size (32MB in this benchmark, and even larger sizes are used in Hadoop such as 64MB and 128MB). It is well known that large chunks are used in existing MR systems to reduce per-chunk overhead, but they hurt latency. Hence, stream systems should consider a smaller chunk size as the granularity of scheduling, without incurring high per-chunk overhead.

(3) *Latency Caused by Disk Spills.* We next consider the case that the aggregate memory in reducers is insufficient to hold all key-state pairs. We use trigram counting (type-1 workload) over the Twitter dataset to show the observations as it has the largest key-state space, e.g., 4-5 times of the aggregate memory of reducers. As Figure 5.1(b) shows, the latency in the Reduce Wait phase of Incremental Hadoop becomes significantly higher (70 sec) than that in the sufficient memory case, but less than sort-merge. This is because

Incremental Hadoop performs incremental updates only to the keys held in-memory. The tuples spilled to disk are processed at the end of the job and hence suffer from high RW. To reduce such latency, a better scheduling strategy is needed to determine the order to process tuples.

Summary. The above observations were also made when we ran type-2 workloads involving UDFs. These observations suggest that to enable streaming analytics with bounded latency of processing (e.g., 1 second) through a distributed system: (1) A new architecture is needed to create and process mini-batches of streaming data without incurring high overhead. (2) It is crucial to determine the *degree of parallelism* (e.g., the number of mappers/reducers per node) and *granularity of scheduling* (e.g., batching data items every 5 ms for shuffling). (3) Under constrained memory, a latency-aware scheduling strategy is needed to determine which tuples to process and in what order to process them. We address these issues in the next three sections.

5.2 System Design

Based on the insights learned from the benchmark study, we now propose a series of architectural changes to transform Incremental Hadoop to a new platform for "fast data" processing. We also outline the overall system design, which provides a technical context for our discussion in the following sections.

5.2.1 Overview of A Scalable Stream System

At a high level, an analytical query in our scalable stream system is modeled as a direct acyclic graph (DAG) of computation units. A computation unit is a pair of map() and reduce(), called an MR-pair. Similar to the traditional MapReduce model, map() is a stateless operation, usually used to extract and filter tuples, while reduce() is a stateful

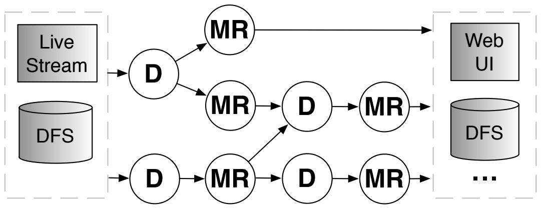


Figure 5.2. A query modeled as a dataflow DAG.

operation used to perform analytics over a group of tuples of the same key. In an MR-pair, map() can be an empty, or reduce() can be empty, but not both.[2]

More precisely, a query in our system is defined as a dataflow DAG, as shown in Figure 5.2. A vertex in the DAG is either a data distributer ("D" vertex) or an MR-pair ("MR" vertex), and an edge represents a stream of tuples flowing between the vertexes. Each tuple is encoded as a triplet $\langle$timestamp, key, value$\rangle$. A distributor can take one or multiple streams of tuples from external sources or upstream MR-pairs, as well as files from a distributed file system (DFS). It feeds the received tuples to one or multiple downstream MR-pairs. An MR-pair takes an input stream from a distributor, performs computation over the stream, and outputs a stream to web UI, DFS, or one or multiple distributors.

Incremental Updates. Our system provides a low-level API to program an MR-pair. Like before, a `map` function is applied to transform each input tuple (a triplet here) to a list of output tuples.

```
map(time,key1,val1) → list(time,key2,val2)
```

where all the arguments here indicate data types. For reduce processing, two functions are used:

```
init(key2) → state
update(time,key2,val2,state) → (state,list(time,key3,val3))
```

[2]How to compile a query into a DAG of MR-pairs is within the purview of MapReduce query compilers such as PigLatin [34], while in this thesis we focus on system support to run a given query plan with low latency.

Reduce is stateful, and a computation state is maintained for each map output key. `init` is called to create an empty state when a new key is received from map output. `update` is triggered by each tuple received, which takes the unique state for a given key and updates it using the tuple. The `update` function can also emit a list of output tuples. This programming model is similar to those in [14] and [55], and has been used in a range of real-world applications.

Time windows. Our work also provides an API for time window based operations. A query defines time windows by specifying the range r and slide s. Given a system starting time t_0, the $\langle r, s \rangle$ pair defines a series of time windows, $(t_0 + i \cdot s,\, t_0 + r + i \cdot s]$, where $i = 0, 1, \cdots$. These windows can be **tumbling** (non-overlaping) or **sliding** (overlapping) windows. After a query is compiled, those time windows that overlap with the lifetime of the query will trigger actual processing. More specifically, the map API remains unchanged, and the reduce API consists of the following three functions:

```
init(key2,wtime) → state
update(time,key2,wtime,val2,state) → state
finalize(key2,wtime,state) → list(time,key3,val3)
```

Now the system maintains a state for each combination of key and time window. Each time window can be identified by its end time, denoted by *wtime*. Hence, the ⟨key, wtime⟩ pair, called a *partitioned window*, indicates a unique instance of windowed operation. For each partitioned window, `init` is called when a new key is seen from map output, and `update` is triggered by each arriving tuple, which takes the state for the relevant partitioned window and updates it with the tuple. Since windows may overlap in time, a tuple will trigger `update` for all relevant ⟨key, wtime⟩ pairs. Finally, `finalize` is called to complete the computation of a partitioned window and generate output tuples.

We now show how to use our API to implement query workloads, including examples of type 2 and type 3 workloads listed in Table 5.1.

Example 5.2.1 (Incremental Updates with UDF – Type 2)**.** *Consider a click stream with attributes (time, userId, url). For each click, determine if the click starts a new session of a user based on user-defined criteria. If so, close the previous session and output it.*

This workload can be coded using the API for incremental updates:

```
map(Time time, SrcID key, Click value):
  // input: click time, resource id, click record
  Emit(time,value.userId(),value);

init(UserID key):
  return new Session();

update(Time time, UserID key, Click value,
       Session state):
  // input: click time, user id, a new click,
  //        the existing session as the state
  if (state.isNewSession(time,value))
    Emit(time, key, state);
  state.add(time,value);
```

For each click, `map` emits the time of the click as timestamp, user id as key, and the click as value. `init` creates an empty state of a user-defined type, `Session`, for each user id. `update` checks whether a click starts a new session of a user using a UDF. If so, it outputs a triplet including the time of the click, the user id, and the entire session. Otherwise, the state is extended with the new click.

Example 5.2.2 (Windowed Aggregates – Type 3)**.** *Return the words that occur more than 100 times in the past 30 seconds from the Twitter firehose. Output every 30 seconds.*

This query uses 30 second tumbling windows. The range and slide of windows are first set for an MR-pair before the job starts.

```
  mr.setWindowRange(30); mr.setWindowSlide(30);
```

Then the MR-pair can be implemented using the windowed API:

```
map(Time time, SrcID key, String value):
  // input: tweet time, resource id, tweet
  for each word w in value: Emit(time,w,1);

init(String key, Time wtime):
  // input: word, end time of a window
  return new Long(0);
```

```
update(Time time, String key, Time wtime,
       Long value, Long state):
  // input: tweet time, word, end time of the window,
  //        constant 1, work count in the window
  state = state + value;

finalize(String key, Time wtime, Long state):
  // input: word, end time of the window,word count
  if (state > 100) Emit(wtime, key, state);
```

Here `init` initializes a counter for each partitioned window, i.e., a (word, wtime) pair, if the word occurs in the specified time period. `update` increments the counter by 1 for each occurrence of the word in the partitioned window. `finalize` checks whether the counter of each partitioned window is over 100 and if so, emits a new tuple with the window end time, key, and counter value.

When our system (which is a runtime system) is integrated with a MapReduce query compiler, such as Pig/Latin and Hive, a high-level SQL-based stream query can be compiled into a DAG query plan in our system. For example, the following stream query will be transformed to a single MR-pair as described in Example 5.2.2.

```
SELECT word, COUNT(*)
FROM FlatternedTweets [Range 30 sec, Slide 30 sec]
GROUP BY word
HAVING COUNT(*) > 100
```

5.2.2 Extended MapReduce Architecture

We next propose necessary changes of the MapReduce architecture to support stream queries with low-latency. We explain these design differences using a single MR-pair as shown in Figure 5.3.

Handling stream data using mini-batches and queues. The first set of changes is proposed to break stream input into mini batches and process them with low overhead, including the use of data distributors, queues, and long-living mappers.

We add a distributor (D) that can take streaming input from an external data source or an upstream MR-pair. For an external data source, the distributor tags each input tuple with a

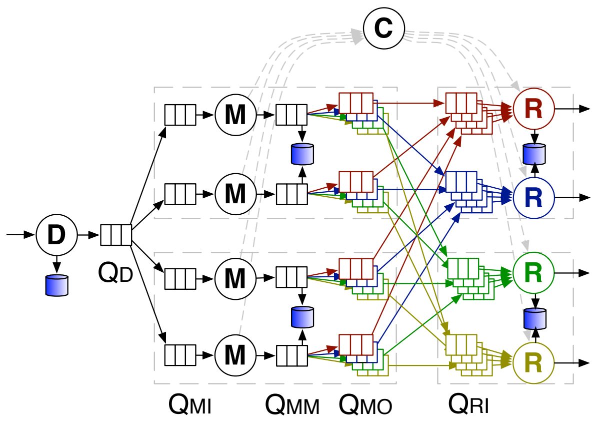


Figure 5.3. MapReduce architecture extended with distributor & queues of mini batches.

system timestamp. The distributor packs the input tuples into a mini batch every B_{in} seconds, as opposed to large batch, to reduce the delay of tuples at the distributor. It places the batch in an in-memory queue, $\mathrm{Q_D}$, such that mappers can fetch data from memory without I/O overhead. The distributor can optionally materialize the mini batches to a DFS for fault tolerance. The distributor can run in multiple processes and nodes to avoid becoming the bottleneck of the system.

We modify a mapper (M) to be able to live forever, rather than terminate after processing a batch. Thus, we can avoid high mapper startup cost caused by each mini batch. We also add several in-memory queues to each mapper. A mapper requests input batches from the distributor, and places the fetched batches in the input queue $\mathrm{Q_{MI}}$. It then applies the map function to each input tuple from $\mathrm{Q_{MI}}$, and emits intermediate tuples, which are further packed into shuffle mini batches every B_{sh} seconds. A shuffle batch is added to the queue $\mathrm{Q_{MM}}$, where it is materialized for fault tolerance, and then split into partitions and placed into the queues $\mathrm{Q_{MO}}$ corresponding to different reducers to send. The mapper then informs the coordinator (C) of the availability of each shuffle batch.

On the reducer side, we add an in-memory queue $\mathrm{Q_{RI}}$ to a reducer (R). A reducer asks the coordinator for the available shuffle batches every B_{ch} seconds, then fetches the batches, and places them in $\mathrm{Q_{RI}}$. We also add a key-value store (currently implemented using BerkeleyDB) to store key-state pairs when memory is not enough.

Handling out-of-order data. Since windowed operations are *order-sensitive*, the system must handle out of order data due to delayed network delivery and different progress of workers in cluster computing. In particular, even if all the tuples arrive from external sources in time order or they are simply timestamped by the system in arrival order, after they are partitioned and assigned to different mappers and processed on different nodes, there is no guarantee that the tuples with the same key will arrive at a reducer in time order. Since each reducer needs to identify the complete set of tuples that fall in a time window, the system must have a way to inform the reducer whether it has delivered all the tuples in a time window.

There are two common solutions in the literature: One is to use punctuations [57] or low watermarks [4], which announce that at time t, an operator has seen all the data up to time $t - \delta$. If a reducer receives such an announcement, it can process all the windows whose end times precede $t - \delta$. However, the most recent study on the MillWheel system [4] reports that out-of-order data can prevent low watermarks from advancing for large amounts of time.

The most advanced technique for out-of-order data [53] proposes to process significantly late data separately in order to reduce latency and buffering cost without dropping data. We adapt this idea in our system as follows. Consider our example of 30-second windows, and assume that time is marked using integers 1, 2, 3, ... Tuples that arrive before time 30 belong to the window $W_{30}^{v_1}$, where 30 is the end time and v_1 is the version number. At time 30, the window closes and a result of $W_{30}^{v_1}$ is ready for output (or buffering). For the late tuples that arrive between time 30 and 60 but should belong to W_{30}, we initiate a new window $W_{30}^{v_2}$ (which is possible because the reduce task permits incremental updates). At time 60, a result of $W_{30}^{v_2}$ is ready for output (or buffering), and so on. At the same time, the MR coordinator asks mappers to report the range of timestamps that they have processed recently. The coordinator can use this information to estimate how many late tuples have not been shuffled to reducers, which can be communicated to the reducers to estimate the "completeness" of the current result of each window. Then the system can be configured to output results of a window only when completeness reaches 100%, which is the same as the low watermark mechanism, or to output results early together with the estimated measure for completeness.

Fault tolerance. Finally, fault tolerance has been intensively studied in MapReduce-style stream systems such as MillWheel [4], Spark Streaming [99], and Storm [92]. Since it is an issue orthogonal to our goal of analyzing latency, we simply adopt the fault-tolerance mechanism of MillWheel [4]. This mechanism combines upstream backup, which writes mini-batches to disk using synchronous I/O before sending them to downstream operators, and checkpointing, which writes the states of windows to disk periodically to reduce the recovery cost. These costs are included in our analysis below.

Maintaining states. For the API for incremental updates, a reducer maintains a state for each key in a hash table. At the execution time of each tuple in a reducer, the reducer looks up the state based on the key in the hash table, and executes the `update` function. For the API for time windows, a reducer maintains a state for each $\langle$key, wtime$\rangle$ pair. We maintain a hash table for each distinct wtime, which organizes the key-state pairs associated with the wtime. For tumbling windows, a tuple falls in exactly one window, and for sliding windows, a tuple may fall in multiple windows. At the execution time of each tuple, the reducer first identifies all the wtimes of the windows the tuple falls in, then looks up the states based on the key in all the hash tables of the wtimes, and finally executes the `update` function over all the retrieved states. After the execution of the `finalize` function for a wtime, we remove the corresponding hash table to save memory. For both APIs, when memory is not enough, we add a key-value store (currently implemented using BerkeleyDB) in reducers, and govern the paging of key-state pairs between memory and the key-value store with DINC-hash (Section 4.1.3), which keeps the frequent keys in memory to minimize disk I/O.

Table 5.2. System parameters in modeling.

Symbol	Description
S	The number of slave nodes
M	The number of mappers per node
R	The number of reducers per node
B_{in}	The distributer packs input tuples into an input batch every B_{in} seconds
B_{sh}	A mapper packs map output tuples into a shuffle batch every B_{sh} seconds
B_{ch}	A reducer checks the available shuffle batches from each mapper every B_{ch} seconds

5.3 Resource Planning

Our previous benchmark study indicates that to enable streaming analytics with bounded latency of processing, it is crucial to determine the *degree of parallelism* (e.g., the number of mappers/reducers per node) and *granularity of scheduling* (e.g., batching data items every 5 ms for shuffling). The appropriate choices of those parameters vary widely among analytic tasks due to different computation needs – using the fixed values tuned for one workload is far from ideal for other workloads, often resulting in high latency of tuples moving through the system. We refer to this problem as *job-specific resource planning*.

To offer best usability, in this section we propose a model-driven approach to automatically determining the resource allocation plan for each job. Below, we explain how our approach addresses performance (latency and throughput) in a holistic manner and supports a variety of latency models, including per-tuple latency, per-window latency, and any quantiles associated with these latency distributions.

5.3.1 Model-driven Resource Planning

Given the job of an analytical query, an estimated data input rate λ_0, and a latency constraint $\mathcal{L}$, our goal is to find the optimal resource allocation plan for the job. To do so, we start by considering the relationship among latency, throughput, and resources. A naive approach may minimize latency by giving all resources to push one tuple at a time through the distributed system, which limits throughput severely. Instead, we aim to support both latency and throughput by taking latency as constraint and maximizing throughput under this constraint. To further take resources into account, we consider a cluster of $\bar{S}$ available slave nodes. For each number of slave nodes $S \in \{1, \ldots, \bar{S}\}$, we compute the maximum input rate, $\Lambda_{\mathcal{L}}(S)$, that can be sustained by S nodes under $\mathcal{L}$, as well as the optimal setting of key system parameters, denoted as $\Theta_{\mathcal{L}}(S)$, that reaches the maximum input rate. Then, to configure the system for *"a job with input rate λ_0 and latency constraint $\mathcal{L}$"*, the smallest value of S that satisfies $\Lambda_{\mathcal{L}}(S) \geq \lambda_0$, denoted by S^*, is the minimum number of nodes to use, and $\Theta_{\mathcal{L}}(S^*)$ is the optimal configuration for the S^* nodes. Finally, our approach returns $(S^*, \Theta_{\mathcal{L}}(S^*))$ as the **resource allocation plan** for the analytic job.

In this approach, a key task is to find the maximum input rate that can be sustained (throughput), $\Lambda_{\mathcal{L}}(S)$, and the optimal setting of parameters, $\Theta_{\mathcal{L}}(S)$, for each $S \in \{1, \ldots, \bar{S}\}$. Let λ denote an input rate in number of tuples per second, $\boldsymbol{\theta}$ be a vector of key system parameters, and $\Psi_{\lambda,(S,\boldsymbol{\theta})}$ be the latency[3] under the input rate λ and resource allocation plan $(S, \boldsymbol{\theta})$. Then, we obtain $\Lambda_{\mathcal{L}}(S)$ and $\Theta_{\mathcal{L}}(S)$ by solving a constrained optimization problem for each S as follows:

$$\begin{aligned} \Lambda_{\mathcal{L}}(S) &= \mathop{\mathbf{max}}_{\boldsymbol{\theta}} \ \lambda, \qquad \textbf{subject to} \ \ \Psi_{\lambda,(S,\boldsymbol{\theta})} \leq \mathcal{L}; \\ \Theta_{\mathcal{L}}(S) &= \mathop{\mathbf{arg\,max}}_{\boldsymbol{\theta}} \ \lambda, \qquad \textbf{subject to} \ \ \Psi_{\lambda,(S,\boldsymbol{\theta})} \leq \mathcal{L}. \end{aligned} \tag{5.1}$$

At the core of our approach is the analytical model of $\Psi_{\lambda,(S,\boldsymbol{\theta})}$, which is built on λ and $(S, \boldsymbol{\theta})$, as well as job-specific characteristics and hardware specification of the cluster. Table 5.2 summarizes all the parameters in $(S, \boldsymbol{\theta})$ for each MR-pair (a computation unit as defined in Section 5.2.1). More specifically, θ includes the numbers of mappers and reducers per node, and the mini-batch sizes of various queues placed in our architecture. While the hardware specification of the cluster can be obtained once for all analytical jobs, job-specific characteristics are provided by the programmer or learned at runtime by the system. (We

[3]The metric can be the average latency or a quantile of latency.

will explain these characteristics more when presenting the detailed models.) Then with the model, $\Psi_{\lambda,(S,\theta)}$, Equation 5.1 can be solved by a general non-linear constrained optimization solver, such as MinConNLP in JMSL[4].

To develop accurate latency models, we identify the major challenges as follows: (1) **Dominant components of latency**: A thorough understanding of the system is required to identify all possible dominant components that contribute to latency. Existing models of MapReduce jobs [40, 41, 45, 30] are designed to predict the running time of a job on stored data. They are not suitable for latency analysis because they ignore key parameters such as the input rate and queue sizes, and factors such as queuing delay and wait time to create a mini-batch, which affect latency strongly. (2) **Shared resources**: Each latency component has to be modeled in a complex environment where system resources are shared by different system modules. Concurrent execution of the map, shuffle and reduce phases are necessary for incremental processing and minimizing latency of output tuples. Existing models [40, 45], however, assume that the map, shuffle and reduce phases do not run in parallel. (3) **Diverse models based on simple statistics**: To offer high usability, it is desirable to support a diverse set of latency metrics, including per-tuple latency, per-window latency, and different quantiles of these latency distributions, while using only the basic job statistics that can be easily provided by the programmer or learned at runtime. None of the existing models can support these latency metrics.

To address these challenges, we choose to model the mean and variance of per-tuple and per-window latency because they enable us to build more complex models for quantiles of latency, while allowing a clean abstraction of various data processing and system-level behaviors using appropriate statistical tools.

[4]http://www.roguewave.com/products/imsl-numerical-libraries.aspx

5.3.2 (μ, σ^2) of Per-Tuple Latency

We begin with the incremental updates workload as defined in Section 5.2.1. For ease of composition, we first focus on one round of MapReduce computation, i.e., one MR-pair. In incremental updates, a result is output when a map output tuple is processed by the `update` function and triggers the current state to satisfy an output criterion. Therefore, we define tuple latency as follows:

Definition 2 (Tuple Latency)**.** *Consider a map output tuple e. Let e' be the (unique) map input tuple that generates e. The latency $\ddot{L}$ of e is the time difference from the distributor receiving e' to the* `update` *function completing the processing of e.*

We assume that the latencies of all tuples are independent, identically distributed (i.i.d.) random variables. Then the observed latency of each tuple can be viewed as a sample drawn from the same underlying tuple latency distribution, denoted as $f_{\ddot{L}}(l)$. Our goal is to model $E(\ddot{L})$ and $Var(\ddot{L})$.

Latency components: With a detailed analysis of the architecture in Figure 5.3, we break $\ddot{L}$ into 12 distinct phases as listed in Table 5.3. Since these phases run sequentially, we have $\ddot{L} = \sum_{i=1}^{12} \ddot{L}_i$. We further assume that $\ddot{L}_i$ and $\ddot{L}_j$ are independent ($i \neq j$). Then

$$E(\ddot{L}) = \sum_{i=1}^{12} E(\ddot{L}_i), \quad Var(\ddot{L}) = \sum_{i=1}^{12} Var(\ddot{L}_i).$$

Thus, we can model $E(\ddot{L}_i)$ and $Var(\ddot{L}_i)$, the mean and variance of latency in each phase, separately.

More fundamentally, we classify these latency components into six types as shown in the last column in Table 5.3: (1) CPU, (2) network, (3) disk I/O, (4) queuing, (5) batching tuples, and (6) heartbeat, i.e, waiting a reducer to ask for new map output. We develop a unified approach to modeling latency types (1), (2) and (3), and show the main principles below. Then we briefly introduce the challenge and our solution to model type (4). Last, we model model (5) and (6).

Table 5.3. Latency breakdown of a map output tuple e (e' is the map input tuple that generates e).

Symbol	Description	Causes of Latency
$\ddot{L}_1$	From when e' reaches the distributor to when e' is packed into an input mini batch.	**Batching**
$\ddot{L}_2$	Queuing delay seen by input batch at queue Q_D prior to network transfer.	**Queuing**
$\ddot{L}_3$	Network latency to transfer the input batch containing e'.	**Network**
$\ddot{L}_4$	Queuing delay seen by the input batch at queue Q_{MI} prior to `map` processing.	**Queuing**
$\ddot{L}_5$	CPU latency needed by `map` function to process the input batch and generate e.	**CPU**
$\ddot{L}_6$	From when e is generated to when e is packed into a shuffle mini batch.	**Batching**
$\ddot{L}_7$	Queuing delay seen by the shuffle batch at queue Q_{MM} prior to disk write.	**Queuing**
$\ddot{L}_8$	Disk latency to write out the shuffle batch containing e.	**Disk I/O**
$\ddot{L}_9$	From when the shuffle batch is added to queue Q_{MO} to when the network begins transferring the batch.	**Queuing+Heartbeat**
$\ddot{L}_{10}$	Network latency to transfer the shuffle batch containing e.	**Network**
$\ddot{L}_{11}$	Queuing delay seen by the shuffle batch at queue Q_{RI}.	**Queuing**
$\ddot{L}_{12}$	CPU latency needed by the `update` function to process the shuffle batch containing e.	**CPU**

CPU, network and disk latencies under shared resources: The latency in this category is determined by the processing time of a batch by the respective resource, i.e. CPU cycles, network bandwidth, or disk bandwidth. $E(\ddot{L}_i)$ in this category can be generally modeled as u/v, where u is the *total* resource required by a batch on average, and v is the resource available to the batch *per second.* (1) **Estimate** u: We estimate u by $m \cdot u_t$, where m is the average number of tuples per batch and u_t is the average resource required per tuple. In general, m can be computed from the data rate and the batch size. Depending on where the batch is in the MR system, the data rate needs to be revised based on the number of mappers or reducers, and input-to-output ratio, α, of the `map` function (in number of tuples) if the batch is downstream of `map`. The statistics required to estimate u, i.e., u_t and α, can be provided by the programmer from historical data or computed by the system from recent batches. (2) **Estimate** v: Due to the nature of incremental processing, the resources on a

Table 5.4. Hardware specification in modeling.

Symbol	Description
n	Num. of CPU cores per node
C	Available CPU cycles of all cores per node in unit time
N	Network bandwidth per node
D	Disk bandwidth per node

compute node are shared by many threads on the node (one thread per mapper/reducer). Hence, v is less than the total resources V available on the node. To address the issue of shared resources, we seek to estimate v using a lower bound by assuming that the other threads have higher priority–such a conservative estimate will make our predicted latency an upper bound of the actual latency, which entails still a valid resource allocation plan through constrained optimization. Our approach is to first estimate p, the fraction of the relevant resource required by all other threads on the same node. Then, we model $v = (1 - p)V$. For $Var(\ddot{L}_i)$ in these types, we directly model them using the sample variance, which is empirically measured from the test runs of the workload.

CPU Latency. The latency components, $\ddot{L}_5$ and $\ddot{L}_{12}$, model the CPU processing latency of the `map` and `update` functions over a batch of tuples, respectively. Since they model the time from the start of the processing of a batch to when a specific tuple in the batch is processed, and a tuple is in a uniformly random position in the batch, $E(\ddot{L}_5)$ and $E(\ddot{L}_{12})$ can be modeled as *half of the processing time of the batch.* Then we model the average processing time l of a batch using the above approach. Here, u is the number of CPU cycles used to process a batch on average, which can be obtained by testing the average CPU cost per tuple and estimating the size of a batch. V is the number of processing cycles that the CPU has per unit time. If we know p, the fraction of CPU cycles consumed by all other threads on the same node, we can model $l = u/((1 - p) \cdot V)$. Further, consider that each CPU has n cores, and a mapper (or reducer) has a single thread to run `map` (or `update`), the fraction of CPU cycles available to the batch is bounded by $1/n$. We then model the average time to process the batch as:

$$E(\ddot{L}_{cpu}) = \frac{u}{2 \cdot \min(1-p, 1/n) \cdot V}. \tag{5.2}$$

To model p, let c_m (c_r) be the CPU cycles required by a mapper (reducer) per unit time, which can be computed from the number of tuples processed by a mapper (reducer) in unit time. The total CPU cycles required per unit time on a node is

$$c_{total} = M \cdot c_m + R \cdot c_r.$$

When we estimate the running time of a batch in a mapper or reducer, we exclude the cost of the current thread itself from c_{total}, and model p as $(c_{total} - c_m)/C$, or $(c_{total} - c_r)/C$. Plugging p into Equation 5.2, we obtain the model of $E(\ddot{L}_{cpu})$.

We model *network* and *disk I/O latency* in a similar fashion.

Network latency. The latency components $\ddot{L}_3$ and $\ddot{L}_{10}$ model the network transmission time of a batch. Suppose the network bandwidth is N, and the average size of a batch is b. Given p, the fraction of bandwidth used by all other network activities on the same node, we can model the average transmission time of the batch as

$$E(\ddot{L}_{net}) = \frac{b}{(1-p) \cdot N}.$$

p can be modeled by considering r, the total size of data transferred in unit time on the same node by all other senders or receivers. Then, we model $p = r/N$.

Disk I/O latency. The latency component $\ddot{L}_8$ models the average time to write a map output batch to local disk. Suppose D is the disk bandwidth on a node, and the average size of a map output batch is b. Given p, the fraction of bandwidth used by all other mappers on the same node, we can model the average time to write a batch as

$$E(\ddot{L}_{disk}) = \frac{b}{(1-p) \cdot D}.$$

p can be modeled by considering d, the total size of data written in unit time on the same node by all other mappers. Then, we model $p = d/D$. In general, the disk bandwidth D is sensitive to q, the number of writes per unit time. Hence, rather than modeling D as a constant, we model D as a function of q, and estimate an empirical model with one-time cost.

Queuing delays have been well studied in the area of queuing theory. The challenge in our problem is to select an accurate model based on the characteristics that can be easily obtained. After surveying of a wide range of queueing models [74, 46, 50, 6], we decide to model each queue as a G/G/1 queue because the more restrictive models, such as M/M/1, make assumptions that are not true in MR systems, while the more general models, such as queuing network, further complicate our model and may require statistics hard to obtain. Queuing theory of the G/G/1 model [50, 6] states that the mean and variance of the queuing delay can be modeled based on the first, second and third moments of T_a and T_s, where T_a and T_s are the random variables for the inter-arrival time between two consecutive batches and the service time of a batch, respectively:

$$E(\ddot{L}_{queue}) = \frac{Var(T_a) + Var(T_s)}{2(E(T_a) - E(T_s))},$$

$$Var(\ddot{L}_{queue}) = E(\ddot{L}^2_{queue}) - E(\ddot{L}_{queue})^2,$$

where

$$E(\ddot{L}^2_{queue}) \approx \frac{E(T_a^2) - 2E(T_a)E(T_s) + E(T_s^2)}{E(T_a) - E(T_s)} \cdot E(\ddot{L}_{queue}) + \frac{E(T_a^3)}{3(E(T_a) - E(T_s))}$$

$$-\frac{E(T_a^3) - 3E(T_a^2)E(T_s) + 3E(T_a)E(T_s^2) - E(T_s^2)}{3(E(T_a) - E(T_s))}.$$

We have modeled $E(T_s)$ for CPU-, network- and disk-type consumers, i.e. $E(\ddot{L}_i)$ in these types. $E(T_a)$ can be computed from the average number of batches added to the queue per unit time. Higher moments of T_a and T_s can be measured empirically.

Batching delay. The latency components $\ddot{L}_1$ and $\ddot{L}_6$ arise when creating batches of tuples. $\ddot{L}_1$ and $\ddot{L}_6$ capture the latency from when a tuple arrives at the batching phase to when a batch is created. Since a batch is created periodically and we assume the arrival time of a tuple to be evenly distributed, we model $E(\ddot{L}_1)$ and $E(\ddot{L}_6)$ as *half of the time between the creation of two consecutive batches*: $E(\ddot{L}_1) = B_{in}/2$ and $E(\ddot{L}_6) = B_{sh}/2$, where parameters B_{in} and B_{sh} denote the periods to create batches in the distributor and a mapper. Also, based on the assumption of even distribution, we model $Var(\ddot{L}_1) = B_{in}^2/12$ and $Var(\ddot{L}_6) = B_{sh}^2/12$.

Heartbeat. A portion of latency component $\ddot{L}_9$ is caused by the gap between two consecutive times a reducer inquires for map output, which is controlled by the parameter B_{ch}. Assuming that the map output batch creation is evenly distributed between two consecutive reduce inquiries, we model the second part of $E(\ddot{L}_9)$ by $B_{ch}/2$ and the second part of $Var(\ddot{L}_9)$ by $B_{ch}^2/12$.

5.3.3 (μ, σ^2) of Per-Window Latency

Unlike incremental update workloads, the results in the windowed workloads are computed only after a reducer has received **all** the tuples in a time window. Therefore, the per-tuple latency cannot reflect the latency of a windowed result. In this section, we define and model the latency of a windowed result. Recall from Section 5.2.1 that a window query defines a series of time windows, $(t_0 + i \cdot s, t_0 + r + i \cdot s]$, where r is the window size, s is the slide, and $i = 0, 1, \ldots$ For each time window, tuples can be further partitioned by the key, resulting in a set of *partitioned windows*, each of which produces a unique windowed result (if non-empty).

Definition 3 (Window Latency)**.** *The latency $\widetilde{L}$ of a partitioned window, denoted as $\langle$key,window$\rangle$, is the time difference from the end-time of the window to the point that the* `finalize` *function completes the processing of $\langle$key,window$\rangle$.*

We assume that latencies of all partitioned windows are i.i.d random variables. Then, the observed latency of each partitioned window can be viewed as a sample from the same

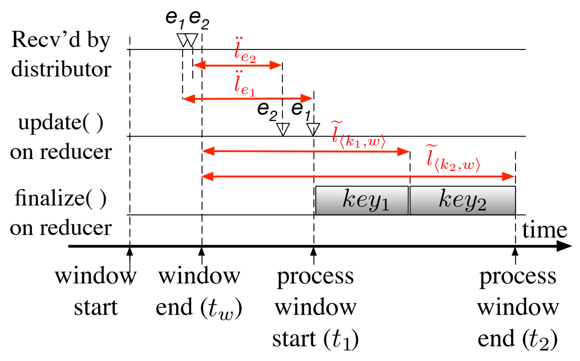


Figure 5.4. An example of window latencies.

window latency distribution, denoted as $f_{\widetilde{L}}(l)$, and our goal is to model $E(\widetilde{L})$ and $Var(\widetilde{L})$. As we shall show shortly, a new major latency component of $\widetilde{L}$ is the *delay of the last tuple of the partitioned window* received by a reducer. The challenge is to choose appropriate tools to model such delay based on easy-to-obtain statistical measurements.

Latency components. Our key observation of the window processing behavior is that once the wall-clock time reaches the end-time t_w of a window, all of its partitioned windows are processed at the same time in two phases: In Phase 1, the system executes the `update` function until all tuples with timestamps earlier than t_w are processed. We denote the completion time of this phase as t_1. Then, in Phase 2, the system executes the `finalize` function over all the related non-empty partitioned windows and completes at t_2.

Figure 5.4 shows the t_w, t_1 and t_2 of an example window in a reducer, with input from two mappers. Tuple e_1 is the last tuple of the window received by the reducer from mapper 1, e_2 is the last tuple from mapper 2, and they correspond to the two distinct keys in the window. At time t_1, when both e_1 and e_2 have been processed by `update` (), the reducer executes `finalize`() on the non-empty partitioned windows corresponding to key_1 and key_2 sequentially. The observed latencies of the two partitioned windows are marked as $\widetilde{L}_{\langle key_1,w\rangle}$ and $\widetilde{L}_{\langle key_2,w\rangle}$, both of which contain $(t_1 - t_w)$ and a portion of $(t_2 - t_1)$ determined by the completion time of the corresponding `finalize` execution. Denote the observed tuple latencies of e_1 and e_2 by $\ddot{L}_{e_1}$ and $\ddot{L}_{e_2}$. We estimate $(t_1 - t_w)$ with

$\max(\ddot{L}_{e_1}, \ddot{L}_{e_2})$. Then, we can approximate: $\widetilde{L}_{\langle key_1,w\rangle} \approx \max(\ddot{L}_{e_1}, \ddot{L}_{e_2}) + (t_2 - t_1)/2$, and $\widetilde{L}_{\langle key_2,w\rangle} \approx \max(\ddot{L}_{e_1}, \ddot{L}_{e_2}) + (t_2 - t_1)$.

Model. Formally, the quantity $(t_1 - t_w)$ is uncertain, and we model it using a random variable U. Let t' denote the time when the `finalize` function completes the execution of a particular partitioned window ($t_1 \leq t' \leq t_2$). Since the quantity $(t' - t_1)$ is also uncertain, we model it using a random variable F. It is easy to get $\widetilde{L} = U + F$. Assume that U and F are independent. Then,

$$E(\widetilde{L}) = E(U) + E(F), \quad Var(\widetilde{L}) = Var(U) + Var(F).$$

We only need to model the mean and variance of U and F separately.

Mean and Variance of U. In the above example, $\widetilde{L}_{\langle key_1,w\rangle}$ and $\widetilde{L}_{\langle key_2,w\rangle}$ are two samples drawn from the distribution of $\widetilde{L}$. Based on the insights from the above example, we model

$$U = \max(\ddot{L}_{e_1}, \cdots, \ddot{L}_{e_{S\cdot M}}),$$

where e_i is the last tuple of the window received by the reducer from mapper i. The moments of $\max(\ddot{L}_{e_1}, \cdots, \ddot{L}_{e_{S\cdot M}})$ are studied in order statistics [24]. We have assumed that $\ddot{L}_{e_1}, \cdots, \ddot{L}_{e_{S\cdot M}}, \ddot{L}$ are i.i.d. in Section 5.3.2. Since we observe the distribution of $\ddot{L}$ to be well centered at its mean in most workloads, we assume that $\ddot{L}$ follows a Gaussian distribution $\mathcal{N}(\ddot{\mu}, \ddot{\sigma}^2)$. Based on Fisher-Tippett-Gnedenko theorem, the following approximation can be applied [24]:

$$\begin{aligned} E(\max(\ddot{L}_{e_1}, \cdots, \ddot{L}_{e_{S\cdot M}})) &\approx \ddot{\mu} + a \cdot \ddot{\sigma}, \\ Var(\max(\ddot{L}_{e_1}, \cdots, \ddot{L}_{e_{S\cdot M}})) &\approx (a' \cdot \ddot{\sigma})^2, \end{aligned}$$

where a and a' are constants relying only in the total number of mappers $(S \cdot M)$:

$$a = \gamma \Phi^{-1}\left(1 - \frac{e^{-1}}{S \cdot M}\right) - (\gamma - 1)\Phi^{-1}\left(1 - \frac{1}{S \cdot M}\right),$$

$$a' = \frac{\pi}{\sqrt{6}} \left(\Phi^{-1} \left(1 - \frac{e^{-1}}{S \cdot M} \right) - \Phi^{-1} \left(1 - \frac{1}{S \cdot M} \right) \right).$$

We can see that $\ddot{\mu}= E(\ddot{L})$ and $\ddot{\sigma}^2= Var(\ddot{L})$ are required, which we have modeled in Section 5.3.2.

Mean and Variance of F. Assume that each time window has the k non-empty partitioned windows on average, and s_k is the CPU cycles required to process a partitioned window by `finalize`(). k and s_k can be estimated by the average measured empirically. Then, we model $(t_2 - t_1) = (k \cdot s_k)/v$, where v is the CPU resources available to a reducer, which has been modeled in Section 5.3.2. As shown in the above example, for a time window, $(t' - t_1)$ of the ith partitioned window processed by `finalize`() can be modeled with $i \cdot (t_2 - t_1)/k$. Treating $(t' - t_1)$ for all i values as samples of F, we use the sample mean $(1 + 1/k)(t_2 - t_1)/2$ and sample variance $(1 + 1/k)(t_2 - t_1)^2/12$ to model $E(F)$ and $Var(F)$, respectively.

5.3.4 Quantiles of Latency

To simplify notation, we use L to generally refer to $\ddot{L}$ if the workload is an incremental update, or $\widetilde{L}$ if it is a windowed workload. We have modeled μ and σ^2 of L in both types of workloads. Now we model any quantile of L, based on μ and σ^2. Let $Q(x)$ denote the x-quantile of L, which has the following property:

$$Pr\left(L \leq Q(x)\right) = x. \tag{5.3}$$

When the distribution of L is known or can be well approximated, we can model $Q(x) = F^{-1}(x)$, where F^{-1} is the inverse CDF of L, and F^{-1} can be expressed using μ and σ^2 in many cases. For example, when the distribution of L can be approximated by a normal distribution, we can model $Q(x) = \mu + \sigma \cdot \Phi^{-1}(x)$, where $\Phi^{-1}(x)$ is the inverse CDF of the standard normal distribution. When the distribution of L is not observed, we use an

Table 5.5. Job-specific statistics required by the model.

Symbol	**Description** (**Required only in a windowed workload.*)
α	Avg. input-to-output ratio of map() in num. of tuples
b_m, b_r	Avg. num. of bytes per tuple in map() or update() input
s_m, s_r	Avg. CPU cycles req. to proc. a tuple in map() or update()
s_k	Avg. CPU cycles req. to proc. a $\langle key, win \rangle$ in finalize()
$E(T_a^2), E(T_a^3)$	2nd, 3rd moments of batch inter-arrival time in each queue
$E(T_s^2), E(T_s^3)$	2nd, 3rd moments of service time of a batch in each queue
k	*Avg num of non-empty partitioned windows of a window

upper bound of the quantile to model $Q(x)$ in order to provision enough resources to meet the latency constraint. According to Cantelli's inequality,

$$Pr\left(L \leq \mu + \sqrt{\frac{x}{1-x}} \cdot \sigma\right) \geq x. \tag{5.4}$$

According to Equation 5.3 and 5.4, we can have $\mu + \sqrt{\frac{x}{1-x}} \cdot \sigma$ as an upper bound of $Q(x)$. Therefore, we model $Q(x) = \mu + \sqrt{\frac{x}{1-x}} \cdot \sigma$.

5.3.5 Job-Specific Statistics

Finally, we summarize all the job-specific statistics used in our model in Table 5.5, which can be measured with test runs or at runtime. α, b_m, b_r and s_m can be obtained by tracking the following metrics of a mapper: the number and size of input tuples, the number and size of output tuples, and the number of consumed CPU cycles[5]. s_r can be measured by tracking the number of input tuples and the consumed CPU cycles of a reducer during the execution of the `update` function. k and s_k can be measured by tracking the number of non-empty partitioned windows and the consumed CPU cycles of a reducer during the execution of the `finalize` function. $E(T_a^2)$, $E(T_a^3)$, $E(T_s^2)$ and $E(T_s^3)$ could be measured at each queue when batches are added or removed from the queue. All the above metrics can be collected in a period of time to obtain stable numbers.

[5]A mapper or a reducer is a process. Consumed CPU cycles of a process can be collected from a profiler such as the `ps` command.

5.3.6 Supporting Multiple MapReduce Rounds

Now, we consider a query defined as a DAG of MR-pairs as we described in Section 5.2.1. Given the latency constraint $\mathcal{L}$ for the query, we first break it down to a series of constraints $\mathcal{L}_i$ for each MR-pair i, such that on every path p from an external data source to an external data consumer,

$$\sum_{\text{MR-pair } i \text{ is on } p} \mathcal{L}_i \leq \mathcal{L}.$$

Then, we solve the resource planning for each MR-pair i with latency constraint $\mathcal{L}_i$ as described earlier in this chapter, and allocate each MR-pair on a separate set of nodes[6].

To assign each $\mathcal{L}_i$, the difference in the costs of MR-pairs should be considered. To satisfy the job-wise latency constraint $\mathcal{L}$, assigning a tight constraint to a costly MR-pair while assigning a lose constraint on a cheap MR-pair may require a resource allocation plan with a unnecessarily large number of nodes in total. As a heuristic, we assign $\mathcal{L}_i$ proportionally to the CPU cost of MR-pair i,

$$\mathcal{L}_i = c_i \cdot \mathcal{L}',$$

where c_i is the CPU cycles required by MR-pair i under the estimate data input rate, and $\mathcal{L}'$ is a value we will solve shared by all MR-pairs. We also allow hints from users to decide c_i for each MR-pair.

When c_i for each MR-pair is known, we have a constraint for each path p from an external data source to an external data consumer: $\mathcal{L}' \cdot \sum_{i \text{ is on } p} c_i \leq \mathcal{L}$. Since we favor to use fewer number of nodes, and thus to set $\mathcal{L}'$ as lose as possible, we obtain the maximum value of $\mathcal{L}'$ while satisfy all the above constraints:

$$\mathcal{L}' = \min_p \left\{ \frac{\mathcal{L}}{\sum_{\text{MR-pair } i \text{ is on } p} c_i} \right\}.$$

[6]This way, we avoid the interference in resource consumption between different MR-pairs, which may further complicate resource allocation.

5.4 Runtime Scheduling

While our model-based approach to resource planning can reduce latency, it may not always offer optimal performance at time. This occurs if the job-specific characteristics previously provided to the model change at runtime, such as the increase in processing cost due to constrained memory and bursty inputs at runtime. In this section, we propose *runtime scheduling* as an optimization, which selects and prioritizes tuples to be processed by the `update` function in each reducer in order to maximize the total utility gained from such processing. We define the scheduling problem as follows:

Problem Statement. Each tuple e can be represented by $(t_e, \tilde{t}_e, c_e)$, where t_e is the time when the distributor receives the input that generates e, $\tilde{t}_e$ is the time when the reducer receives e, and c_e is the time cost of processing e by the `update` function. The problem is to design an online algorithm for each reducer (i.e. the algorithm knows nothing about e before $\tilde{t}_e$) to decide an order to process tuples sequentially in order to **maximize** $\sum U(e, \hat{t}_e)$, where U is a utility function for tuple e and $\hat{t}_e$ is the time when `update`(e) completes.

We first explain how we obtain the time cost c_e for running `update`() on tuple e. In a reducer, we maintain key-state pairs in an in-memory hash table, where each arriving tuple triggers the update of the computation state for its key. When memory is insufficient to hold all key-state pairs, some of them are staged to a key-value store on local disk (using SSD for better performance). To minimize I/O, we use the FREQUENT algorithm (described in Section 4.1.3) to decide the keys in memory. Then to estimate c_e, we partition tuples into two groups: one group for tuples with keys in memory, and the other for tuples that are staged to disk. Which group a tuple belongs to can be determined by checking the in-memory hash table with low cost. We measure the average per-tuple cost in each group at runtime, and estimate c_e with the average value of the group.

We next present a simple yet popular utility function. Given a user-defined latency constraint $\mathcal{L}$, $U(e, \hat{t}_e) = 1$ if $\hat{t}_e \le t_e + \mathcal{L}$ (deadline of e); $U(e, \hat{t}_e) = 0$, otherwise. That is, if the latency of a tuple is within $\mathcal{L}$, we gain utility 1. Otherwise, we gain 0.

The problem of maximizing total utility in this context is close to online scheduling problems in real-time operating systems [52, 21], where computation tasks are scheduled to maximize utility. The two major differences in our problem are: (1) a tuple represents a task so the number of tasks to schedule is much larger, and (2) processing a tuple is relatively cheap compared to processing a complex task. Therefore, existing scheduling techniques do not fit our problem, due to the high time and space complexity of running the scheduling algorithm for a large number of cheap tuples. To overcome the problems, we propose two scheduling techniques.

5.4.1 Batch-level Scheduling

To reduce scheduling complexity, the first method we propose is batch-level scheduling. It considers a shuffle batch received by a reducer as the scheduling unit. When a batch is scheduled, the tuples in the batch are processed sequentially. The total utility of a batch is the number of tuples in the batch n_b. Regarding the cost of the batch, a key assumption we make is that this cost can be approximated by $n_b \cdot \overline{c_e}$, where $\overline{c_e}$ is the average cost per tuple. This assumption is made to simplify scheduling, but is supported by the following intuition: While the costs of processing individual tuples may vary, when we average them over a batch, the tuple-level differences tend to cancel each other and the average cost can be quite stable, especially if we measure the average cost from recently processed batches. Therefore, we obtain two properties for each batch: (1) The *value density* of a batch, which is the utility divided by the cost, is $1/\overline{c_e}$, a constant across batches. (2) When we process a portion of tuples in the batch, we gain utilities of those tuples, even if we do not complete the batch. Given these two properties, the earliest deadline first (EDF) scheduling is known to be optimal [21].

However, batch-level scheduling processes tuples in a batch regardless of their costs, e.g., spending time to process a tuple with a high cost in a scheduled batch rather than two tuples with low costs in an unscheduled batch. Hence, it may not yield optimal utility.

5.4.2 Tuple-level Cost-aware Scheduling

Next we propose tuple-level cost-aware scheduling, as well as techniques to reduce its scheduling overhead. Tuple-level scheduling does not satisfy the properties of batch-level scheduling. The value density of a tuple varies due to the varying cost, and one only gains the utility of a tuple by fully processing it. Hence, EDF is not suitable for tuple-level scheduling. The D^{over} algorithm [52] achieves the optimal worst-case competitive ratio in this setting. However, it does not consider costs of tuples effectively, and thus can suffer from low utility in practice (verified in Section 5.5), despite the worst-case guarantee in theory.

We propose a new cost-aware scheduling algorithm. To better describe the algorithm, we first define a concept: A set of tuples Γ is **schedulable** at time t if there exists an order to process all tuples in Γ sequentially starting at time t and no tuple misses its deadline. Now we sketch the algorithm, which includes three functions:

- `init()` is called when a reducer is created.
- `release(`Γ'`)` is called when a shuffle batch is received by the reducer. Here the algorithm maintains a schedulable set of tuples Γ to process, which is initially empty. When a set of tuples Γ' are released to the reducer, the scheduler merges Γ' into Γ, finds a largest schedulable subset of Γ at current time, and updates Γ to the schedule subset.
- `nextToProcess()` returns the next tuple in Γ to process and is invoked by the reducer whenever the processing of the previous tuple completes. Since it is known [26] that processing a schedulable set in increasing order of deadline guarantees that no tuple misses its deadline, our algorithm generates a plan to process tuples in Γ in that order. A tuple is removed from Γ immediately when its processing starts.

Sometimes, before all the tuples in the current schedulable set Γ are processed, the reducer can receive a new shuffle batch. Then the algorithm merges the remaining tuples in Γ with the new tuples Γ', finds a new largest schedulable subset from these tuples, and hands it to the reducer for processing through `nextToProcess()`.

Algorithm 3 Sketch of finding a largest schedulable subset.

```
largestSchedulableSubset($\Gamma$)
1: $t \leftarrow$ current time,   Initialize $d_i, \Gamma_i$ from $\Gamma$
2: for $i = 1, 2, \cdots, g$ do
3:     while total cost of $t + \Gamma_1 \cup \Gamma_2 \cup \cdots \cup \Gamma_i > d_i$ do
4:         remove the tuple of the largest cost from $\Gamma_1 \cup \Gamma_2 \cup \cdots \cup \Gamma_i$
5:     end while
6: end for
7: return $\Gamma_1 \cup \Gamma_2 \cup \cdots \cup \Gamma_g$
```

The key part of the algorithm is finding a largest schedulable subset from Γ, which considers both cost and deadline of each tuple. The high-level intuition is that if we enumerate all subsets of Γ, for each subset we can check whether it is a schedulable subset based on the cost and deadline; among those schedulable subsets, we want to find the subset that has the largest number of tuples. Of course, enumerating all subsets is expensive to do. We first outline how we avoid the enumeration in finding a largest schedulable subset, and then introduce an efficient implementation of the scheduling algorithm based on a tree structure. At last, we show the per-tuple scheduling time complexity.

Finding a Largest Schedulable Subset. Let g be the number of distinct deadlines of the tuples in Γ, and $d_1, d_2, \cdots, d_g$ are these distinct deadlines in increasing order. Partitioning Γ based on the deadline gives a tuple set Γ_i for each deadline d_i ($i = 1, \cdots, g$). Algorithm 3 shows the sketch: considering each deadline d_i in increasing order, we remove the tuple with the highest cost in $\Gamma_1 \cup \Gamma_2 \cup \cdots \cup \Gamma_i$ repeatedly, until all remaining tuples in $\Gamma_1 \cup \cdots \cup \Gamma_i$ can be processed before d_i starting at current time t (Lines 2-6). After all deadlines are checked, the remaining tuples in $\Gamma_1 \cup \cdots \cup \Gamma_g$ are returned (Line 7). The following proposition states the correctness of the algorithm.

Proposition 5.4.1. $\Gamma_1 \cup \Gamma_2 \cup \cdots \cup \Gamma_g$ *returned by Algorithm 3 is a largest schedulable subset of* Γ *at time* t.

Before proving Proposition 5.4.1, we prove two lemmas.

Lemma 5.4.1. $\Gamma_1 \cup \cdots \cup \Gamma_g$ *returned by Algorithm 3 is a schedulable subset of* Γ *at time* t.

Proof. If we process the tuples in $\Gamma_1 \cup \Gamma_2 \cup \cdots \cup \Gamma_g$ in increasing order of deadline starting at time t, according to Line 2-4 in Algorithm 3, it is easy to see that $\forall i = 1, 2, \cdots, g$, the completion time of processing tuples in Γ_i is $(t+$ total cost of $\Gamma_1 \cup \Gamma_2 \cup \cdots \cup \Gamma_i) \leq d_i$. That is, no tuple misses its deadline. Thus, $\Gamma_1 \cup \Gamma_2 \cup \cdots \cup \Gamma_g$ is a schedulable subset. □

Suppose Algorithm 3 removes x tuples in total from Γ. Let e_j denote the jth removed tuple ($j = 1, 2, \cdots, x$). Let $\Gamma^{(j)}$ denote the set of remaining tuples after removing e_j. We have $\Gamma^{(0)} = \Gamma$ and $\Gamma^{(x)} = \Gamma_1 \cup \Gamma_2 \cup \cdots \cup \Gamma_g$ returned by Algorithm 3. Note that there may be one or multiple largest schedulable subsets that have equal number of tuples. Let $\mathcal{M}$ be the set of all largest schedulable subsets at time t. The following lemma holds.

Lemma 5.4.2. $\forall j \in \{0, \cdots x\}, \exists \Gamma_{\max} \in \mathcal{M}$ *s.t.* $\Gamma^{(j)} \supseteq \Gamma_{\max}$.

Proof. For $j = 0$, since $\Gamma^{(0)} = \Gamma$ and any tuple set in $\mathcal{M}$ (apparently $\mathcal{M} \neq \emptyset$) is a subset of Γ, it holds that $\exists \Gamma_{\max} \in \mathcal{M}$ such that $\Gamma^{(0)} \supseteq \Gamma_{\max}$.

For $j = 1, 2, \cdots, x$, we prove by contradiction. Assume the lemma does not hold. Consider the smallest value of j s.t. $\forall \Gamma_{\max} \in \mathcal{M}$, $\Gamma^{(j)} \not\supseteq \Gamma_{\max}$. We have $\exists \hat{\Gamma}_{\max} \in \mathcal{M}$ s.t. $\Gamma^{(j-1)} \supseteq \hat{\Gamma}_{\max}$. Since $\Gamma^{(j)} = \Gamma^{(j-1)} - \{e_j\}$, $e_j \in \hat{\Gamma}_{\max}$. Suppose e_j is removed during the check of deadline d_i. According to Algorithm 3, if $\Gamma^{(j-1)}$ is processed in increasing order of deadline, at least one tuple with deadline d_i misses the deadline. Since $\Gamma^{(j-1)} \supseteq \hat{\Gamma}_{\max}$ and $\hat{\Gamma}_{\max}$ is schedulable, $\exists e' \in \Gamma^{(j-1)}$ s.t. the deadline of $e' \leq d_i$, and $e' \notin \hat{\Gamma}_{\max}$. Apparently, $e' \neq e_j$ and the cost of $e' \leq$ the cost of e_j. We construct a tuple set $\hat{\Gamma}_{\max} \cup \{e'\} - \{e_j\}$. It is easy to get that $\hat{\Gamma}_{\max} \cup \{e'\} - \{e_j\}$ is a subset of $\Gamma^{(j)}$. We next prove that $\hat{\Gamma}_{\max} \cup \{e'\} - \{e_j\}$ is a largest schedulable subset of Γ, which contradicts with $\forall \Gamma_{\max} \in \mathcal{M}$, $\Gamma^{(j)} \not\supseteq \Gamma_{\max}$.

In order to prove that $\hat{\Gamma}_{\max} \cup \{e'\} - \{e_j\}$ is a largest schedulable subset of Γ at time t, we first prove $\hat{\Gamma}_{\max} \cup \{e'\} - \{e_j\}$ is schedulable at t by showing no deadline among $d_1, d_2, \cdots, d_g$ is missed if $\hat{\Gamma}_{\max} \cup \{e'\} - \{e_j\}$ is processed in increasing order of deadline, and then prove that $|\hat{\Gamma}_{\max} \cup \{e'\} - \{e_j\}|$ is maximum for a schedulable subset.

We already know that $\Gamma^{(j-1)} \supseteq \hat{\Gamma}_{\max}$ and $e' \in \Gamma^{(j-1)}$.So, $\Gamma^{(j-1)} \supseteq \hat{\Gamma}_{\max} \cup \{e'\} - \{e_j\}$. According to Algorithm 3, if $\Gamma^{(j-1)}$ is processed in increasing order of deadline at time t, no deadline among $d_1, d_2, \cdots, d_{i-1}$ is missed. Thus, if $\hat{\Gamma}_{\max} \cup \{e'\} - \{e_j\}$ is processed in increasing order of deadline, no deadline among $d_1, d_2, \cdots, d_{i-1}$ is missed. Since $\hat{\Gamma}_{\max}$ is schedulable at time t, $\forall k = i, \cdots, g$, ($t+$ total cost of tuples in $\hat{\Gamma}_{\max}$ with deadline $d_1, d_2, \cdots, d_k) \leq d_k$. We know the deadline of $e_j \leq d_i$. We have shown that the deadline of $e' \leq d_i$, and the cost of $e' \leq$ the cost of e_j. Let $\Delta =$ (the cost of $e_j -$ the cost of e'). Apparently, $\Delta \geq 0$. If $\hat{\Gamma}_{\max} \cup \{e'\} - \{e_j\}$ is processed in increasing order of deadline, $\forall k = i, \cdots, g$, the completion time of processing tuples with deadline k is ($t - \Delta +$ total cost of tuples in $\hat{\Gamma}_{\max}$ with deadline $d_1, d_2, \cdots, d_k$)$\leq d_k$. So, no deadline among $d_i, \cdots, d_g$ is missed. To summarize, if $\hat{\Gamma}_{\max} \cup \{e'\} - \{e_j\}$ is processed in increasing order of deadline at time t, no deadline among $d_1, \cdots, d_g$ is missed. Hence, $\hat{\Gamma}_{\max} \cup \{e'\} - \{e_j\}$ is schedulable at time t.

We now prove that $|\hat{\Gamma}_{\max} \cup \{e'\} - \{e_j\}|$ is maximum for a schedulable subset at time t. We have shown that $e_j \in \hat{\Gamma}_{\max}$ and $e' \notin \hat{\Gamma}_{\max}$. So, we have $|\hat{\Gamma}_{\max} \cup \{e'\} - \{e_j\}| = |\hat{\Gamma}_{\max}|$. Since $\hat{\Gamma}_{\max}$ is a largest schedulable subset, $|\hat{\Gamma}_{\max} \cup \{e'\} - \{e_j\}|$ is maximum for a schedulable subset.

In summary, $\Gamma_{\max} \cup \{e'\} - \{e_j\}$ is a largest schedulable subset of Γ, which contradicts with $\forall \Gamma_{\max} \in \mathcal{M}, \Gamma^{(j)} \not\supseteq \Gamma_{\max}$, and hence the lemma holds for $j = 1, 2, \cdots, x$. We have already shown that the lemma holds for $j = 0$. Thus, we have proved the lemma. □

Finally, we prove Proposition 5.4.1 based on the above two lemmas.

Proof. According to Lemma 5.4.2, $\Gamma_1 \cup \cdots \cup \Gamma_g$ returned by Algorithm 3, i.e. $\Gamma^{(x)}$, is a superset of a largest schedulable subset of Γ at time t. According to Lemma 5.4.1, $\Gamma_1 \cup \cdots \cup \Gamma_g$ is schedulable at t. So, $\Gamma_1 \cup \cdots \cup \Gamma_g$ is a largest schedulable subset of Γ at t. □

Operation	Description	Time Complexity
`insert`$(\mathcal{T}, e)$	Insert a tuple e to tree $\mathcal{T}$	$O(\log m)$
`delete`$(\mathcal{T}, e)$	Delete a tuple e from tree $\mathcal{T}$	$O(\log m)$
`schedulable`$(\mathcal{T})$	Return whether all tuples in $\mathcal{T}$ are schedulable now	$O(1)$
`min_unsched_grp`$(\mathcal{T})$	Return minimum d_i such that: (now + total cost of $\Gamma_1 \cup \cdots \cup \Gamma_i) > d_i$	$O(\log g)$
`max_cost_in_grps`$(\mathcal{T}, d_i)$	Return the tuple with the maximum cost in: $\Gamma_1 \cup \cdots \cup \Gamma_i$	$O(\log g)$
`next`$(\mathcal{T})$	Return a tuple with the earliest deadline in $\mathcal{T}$	$O(\log g)$

Table 5.6. Description and time complexity of basic operations over tree $\mathcal{T}$.

A Tree-based Implementation. We now propose an efficient implementation of the tuple-level cost-aware scheduling algorithm using a tree structure $\mathcal{T}$ for organizing tuples in Γ. We implement a few basic operations over $\mathcal{T}$ with corresponding time complexity shown in Table 5.6, where m is the maximum number of tuples in Γ.

More specifically, $\mathcal{T}$ is a balanced binary search tree of all the distinct deadlines in Γ. Let N_i denote the tree node associated deadline d_i. Each node N_i maintains: 1) a max heap $\mathcal{H}_i$ that organizes all tuples in Γ_i based on tuple cost; 2) metadata that summarizes all tuples in the subtree $\mathcal{T}_i$ rooted at N_i, including:

- c_Σ: total cost of tuples in subtree $\mathcal{T}_i$
- $c_{\max}$: the maximum cost of a tuple in $\mathcal{T}_i$
- ϕ: the latest time to start processing all the tuples in $\mathcal{T}_i$ such that the tuples are schedulable

An example of $\mathcal{T}$ with 3 distinct deadlines 10, 15 and 20 is shown in Figure 5.5, where the heap of three tuples with deadline 15 is shown as an example while the other two heaps are omitted.

The tree structure offers two key properties for supporting the operations efficiently. (1) The question, "are the tuples in the subtree $\mathcal{T}_i$ schedulable at time t," can be answered by the boolean expression "$t \leq N_i.\phi$" in $O(1)$ time. (2) The metadata in each node N_i can be

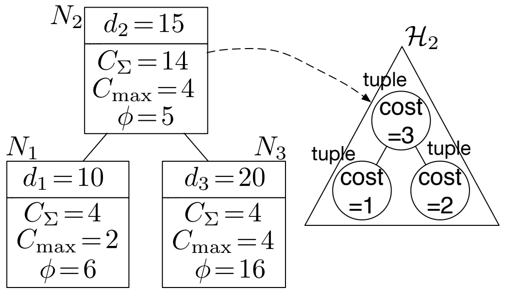


Figure 5.5. An example of tree $\mathcal{T}$.

computed directly from the metadata of its left child N_L, right child N_R, and its associated deadline group Γ_i in $O(1)$ time as follows:

$$\begin{aligned} N_i.c_\Sigma &= N_L.c_\Sigma + N_R.c_\Sigma + \Gamma_i.c_\Sigma \\ N_i.c_{\max} &= \max(N_L.c_{\max},\ N_R.c_{\max},\ \Gamma_i.c_{\max}) \\ N_i.\phi &= \min(N_L.\phi,\ (d_i - N_L.c_\Sigma - \Gamma_i.c_\Sigma),\ (N_R.\phi - N_L.c_\Sigma - \Gamma_i.c_\Sigma)) \end{aligned} \tag{5.5}$$

where $\Gamma_i.c_\Sigma$ is the total tuple cost in Γ_i, and $\Gamma_i.c_{\max}$ is the maximum tuple cost in Γ_i. The calculation of $N_i.c_\Sigma$ and $N_i.c_{\max}$ is straightforward. We explain $N_i.\phi$ more below. Let t be a time when all the tuples in $\mathcal{T}_i$ are schedulable. Processing the tuples in $\mathcal{T}_i$ in increasing order of deadline starting at time t is known to guarantee that no tuple misses its deadline [26]. In such an execution plan, the processing of tuples in the subtree of N_L starts at t, yielding the constraint $t \leq N_L.\phi$. The time when the processing of Γ_i ends, which is also when the processing of the tuples in the subtree of N_R starts, is $t + N_L.c_\Sigma + \Gamma_i.c_\Sigma$, giving the two constraints $t + N_L.c_\Sigma + \Gamma_i.c_\Sigma \leq d_i$ and $t + N_L.c_\Sigma + \Gamma_i.c_\Sigma \leq N_R.\phi$. Combing the three constraints, we have the above equation for $N_i.\phi$. In Figure 5.5, the metadata of N_2 can be computed as described above.

Now we describe in detail the basic operations over $\mathcal{T}$ with corresponding time complexity in Table 5.6.

- ► `insert`$(\mathcal{T}, e)$ involves searching the node associated with the deadline of e in the balanced binary search tree, creating a new node if no corresponding node is found,

Algorithm 4 `min_unsched_grp`$(\mathcal{T})$

```
1: $N_i \leftarrow$ root of $\mathcal{T}$,   $\Phi \leftarrow$ current time
2: while $N_i \neq$ null do
3:     $N_L \leftarrow$ left child of $N_i$,   $N_R \leftarrow$ right child of $N_i$
4:     if $N_L \neq$ null and $\Phi > N_L.\phi$ then
5:         $N_i \leftarrow N_L$
6:     else
7:         $\Phi \leftarrow \Phi + N_L.c_\Sigma$
8:         if $\Phi + \Gamma_i.c_\Sigma > d_i$ then
9:             return $d_i$
10:        else
11:            $\Phi \leftarrow \Phi + \Gamma_i.c_\Sigma$,   $N_i \leftarrow N_R$
12:        end if
13:    end if
14: end while
```

and updating the meta data on the path from the node associated with e to the root. Each of the three steps takes $O(\log g)$ time. The operation also involves inserting e to the corresponding heap, which takes $O(\log m)$ time. Since $g \leq m$, `insert`$(\mathcal{T}, e)$ takes $O(\log m)$ time.

- `delete`$(\mathcal{T}, e)$ involves searching the node associated with the deadline of e, updating the meta data on the path from the node to the root, and deleting the node if the corresponding heap becomes empty. Each of the three steps takes $O(\log g)$ time. The operation also involves deleting e from the corresponding heap, which takes $O(\log m)$ time. Since $g \leq m$, `delete`$(\mathcal{T}, e)$ takes $O(\log m)$ time.

- `schedulable`$(\mathcal{T})$ only compares the current time with ϕ of the root, and thus takes $O(1)$ time.

- `min_unsched_grp`$(\mathcal{T})$ returns the smallest d_i such that $\Gamma_1 \cup \Gamma_2 \cup \cdots \cup \Gamma_i$ is not schedulable at current time. If we know $\Gamma_1 \cup \cdots \cup \Gamma_j$ is schedulable at current time, the problem can be transformed to finding the smallest d_i such that $\Gamma_{j+1} \cup \Gamma_{j+2} \cup \cdots \cup \Gamma_i$ is not schedulable at time Φ, where $\Phi =$current time+total cost of $\Gamma_1 \cup \cdots \cup \Gamma_j$. Based on this observation, we design an algorithm for `min_unsched_grp`$(\mathcal{T})$ that searches top-down in the tree as shown in Algorithm 4. We start from the root node and initialize Φ to current time (Line 1). We first check whether the left subtree is unschedulable (Line 4). If yes, we move to the left subtree for search (Line 5). Otherwise, we update Φ because we know that the left subtree is schedulable (Line 7), and we check whether the deadline group associated with the current tree node is unschedulable (Line 8). If yes, the deadline associated with the current tree node is the desired d_i and we return the value (Line 9). Otherwise, we update Φ since we know the deadline group associated with the current tree node is schedulable, and we move to the right subtree for search (Line 11). We repeat the process (Line 2-11) until we find the desired d_i. Since searching at a tree node takes $O(1)$, and we search at most $O(\log g)$ nodes, the operation takes $O(\log g)$ time.

► `max_cost_in_grps`$(\mathcal{T}, d_i)$ returns the tuple with the maximum cost in $\Gamma_1 \cup \Gamma_2 \cup \cdots \cup \Gamma_i$. First, the algorithm searches top-down to the node N_i associated with deadline d_i. Then, temporarily update the $c_{\max}$ of each node bottom-up on the path from N_i to root in the following way: $c_{\max}$ of N_i is updated to $\max(\Gamma.c_{\max}, N_L.c_{\max})$. For any other node on the path, if N_i is a left descendent of the node, $c_{\max}$ is updated to the same value as its left child; otherwise, compute $c_{\max}$ as usual using Equation 5.5. The next step is searching top-down from the root along a path where all the nodes on the path have the same $c_{\max}$ value as the root, until no child node has the same $c_{\max}$ or the leaf node is reached. The tuple at the top of the heap associated with the last searched node should be returned. The last step is restoring all the temporarily changed $c_{\max}$ values. This operation visits the nodes on a path in the tree four times, and each visit takes $O(1)$ time. Thus, the operation takes $O(\log g)$ time.

► `next`$(\mathcal{T})$ involves searching the node associated with the smallest deadline in $O(\log g)$ time, and retrieving the first tuple from the corresponding heap in $O(1)$ time. So, the operation takes $O(\log g)$ time.

Since `insert`$(\mathcal{T}, e)$ and `delete`$(\mathcal{T}, e)$ may cause insertion or deletion of a node in the tree, they may result in self-balancing of the tree structure. Any rotation-based self-

Algorithm 5 Implement the cost-aware tuple-level scheduling algorithm using the tree structure.

```
init()
1: $\mathcal{T} \leftarrow$ empty tree
release($\Gamma'$)
1: for each $e \in \Gamma'$ do
2:     insert($\mathcal{T}$,$e$)
3: end for
4: while schedulable($\mathcal{T}$) $\neq$ true do
5:     $d \leftarrow$min_unsched_grp($\mathcal{T}$), $e \leftarrow$max_cost_in_grps($\mathcal{T}$,$d$)
6:     delete($\mathcal{T}$,$e$),    Abandon $e$
7: end while
nextToProcess()
1: $e \leftarrow$ next($\mathcal{T}$)
2: if $e \neq$ null then
3:     delete($\mathcal{T}$,$e$)
4:     return $e$
5: end if
```

balancing binary search tree with worst-case complexity of $O(\log g)$ for each insertion and deletion can be used here, such as red-black tree. For such a tree, each insertion or deletion may trigger up to $O(\log g)$ rotations. For each rotation, the meta data of the affected nodes can be updated with $O(1)$ time. Thus, the worst-case complexity for `insert`$(\mathcal{T},e)$ and `delete`$(\mathcal{T},e)$ is not affected.

Finally, based on the tree operations shown in Table 5.6, we implement our tuple-level scheduling algorithm as shown in Algorithm 5, where Lines 4-7 of `release()` reimplement Algorithm 3.

Time Complexity. As shown in Algorithm 5, each of the basic tree operations is called at most once per tuple. Since each operation can be performed in $O(\log m)$ time as we have shown, the amortized time per tuple of the scheduling algorithm is $O(\log m)$, where m is the maximum number of tuples in Γ as described above.

5.4.3 Optimization in Cost-aware Scheduling

In practice, our tuple-level, cost-aware scheduling may still degrade performance due to the $O(\log m)$ complexity per tuple. We next propose several optimizations of our tuple-level scheduling to further reduce the scheduling cost. *First*, we cluster tuples into a fixed number of groups based on similar costs, and estimate the cost of a tuple using the average cost of the group. For example, we described a solution to create a memory group and a disk group earlier in this section. *Second*, we use a configurable coarse-grained time unit for deadlines to lower the scheduling cost by reducing the size of the tree structure. The size of the time unit controls the trade-off between accuracy of deadlines and efficiency of scheduling. As we will show in evaluation, the most utility is gained when the time unit is about one order of magnitude smaller than the latency constraint. *Third*, instead of operating on each individual tuple, a set of tuples from a shuffle batch that share the same cost and deadline can be inserted, deleted and scheduled for processing together by the scheduler. Due to a small number of cost levels and coarse-grained deadlines, a significant number of tuples can be operated together, and thus the amortized per-tuple scheduling cost is reduced.

5.4.4 Extension for Time Windows

Finally, we outline how we adapt the above scheduling techniques to support time windows. The difference with time windows is that in theory, only the output tuples of each window generate utility. However, scheduling needs to be done when tuples arrive at the reducer; at that point, which tuples produce which output is not known. To leverage our previous techniques that requires a utility and a deadline of each tuple, we make several modifications: Assuming that each partitioned window has utility 1, we consider a tuple having partial utility of the window it belongs to, which is 1 over the number of tuples, n_w, in the window. Again, we use the statistics from recently processed windows to estimate n_w. A tuple's deadline is set to the deadline of its window, minus the cost of `finalize()` which performs the final processing when the window closes. Then our scheduling algorithms run as before.

5.5 Performance Evaluation

We have implemented all of our proposed techniques in Incremental Hadoop as described in Chapter 4. In this section, we evaluate the efficiency of our system, with the new modeling and scheduling techniques for reducing latency. We also include a comparison to Spark Streaming [99] and Twitter Storm [92], two state-of-the-art open-source distributed stream systems, which have been widely used in industry.

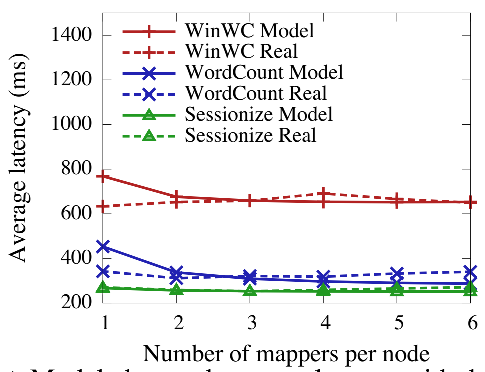
In all of our experiments, we use the same cluster of 10 nodes and same workloads as in our benchmark study in Section 5.1. We show most results from the following workloads: (1) *Word Counting* over the 13GB Twitter dataset (Type 1), (2) *Windowed Word Counting* over the same Twitter dataset using 30-second tumbling time windows (Type 2), and (3) *Sessionization* over the 236GB WorldCup click stream (Type 3). We simulate a streaming data source at a configurable input rate. By default, we configure the system using our model-based resource configuration and allocate sufficient buffer space to mappers and reducers, unless stated otherwise.

5.5.1 Latency-Aware Configuration

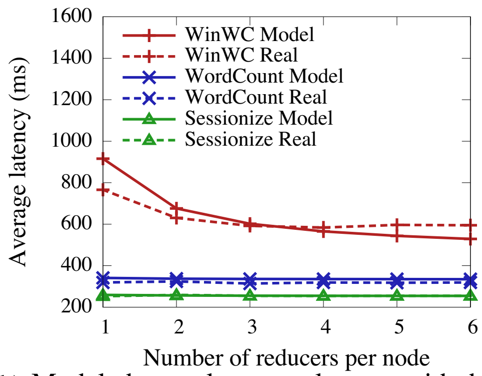
We begin by evaluating the effectiveness of our model-based resource configuration for reducing latency.

Model Accuracy. To evaluate accuracy, we compare our modeled latency with the latency measured in our cluster when tuning five key system parameters. The default setting is: for the numbers of mappers and reducers, $M = R = 2$; for buffer sizes, $B_{in} = 0.01$ sec and $B_{sh} = B_{ch} = 0.2$ sec. The input rate is 0.5 million tweets/sec for the Twitter dataset and 2.5 million clicks/sec for the WorldCup click stream. We consider latency metrics ranging from the average latency to the 0.9- and 0.99-quantiles of latency.

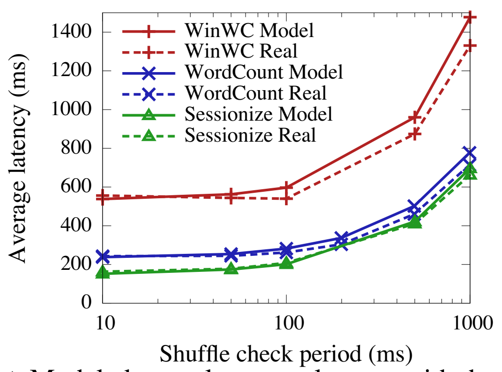
Regarding *average latency*, Figure 5.6(a) shows the results when we tune the number of mappers, M, in the three workloads, while Figure 5.6(b), 5.6(c) and 5.6(d) show those when we tune the number of reducers R, the shuffle check period B_{ch} and the shuffle period B_{sh}



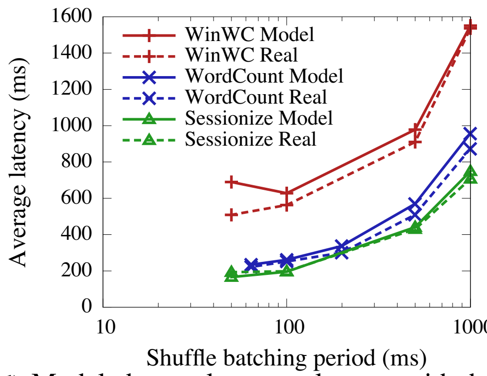
(a) Modeled vs real average latency with the num. of mappers per node, M, tuned.



(b) Modeled vs real average latency with the num. of reducers per node, R, tuned.



(c) Modeled vs real average latency with the shuffle check period, B_{ch}, tuned.

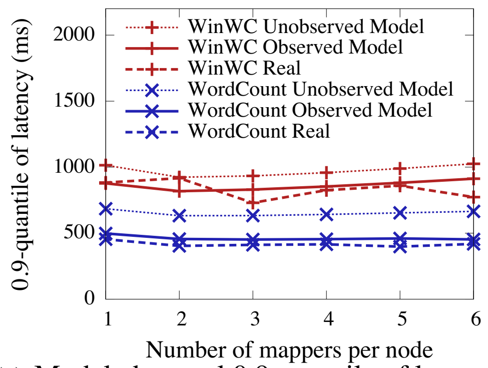


(d) Modeled vs real average latency with the shuffle period, B_{sh}, tuned.

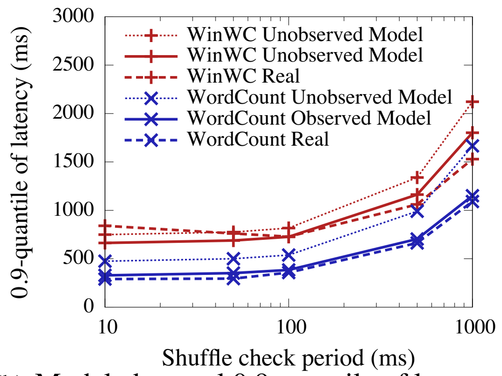
Figure 5.6. Accuracy of the model for average latency.

respectively. The modeled average latency is close to real values in all the workloads. In 60 out of the 65 experiments, the relative error is below 15%.

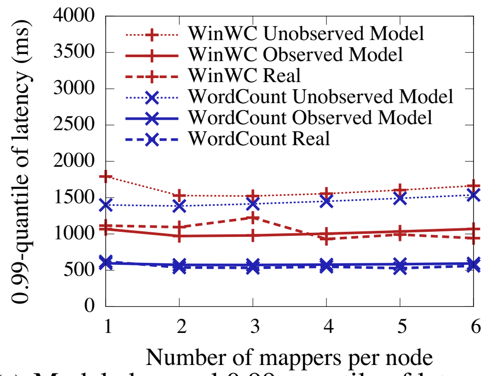
For the *0.9-quantile of latency*, Figure 5.7(a) and 5.7(b) show the results with varied M and B_{ch}, for word counting and windowed word counting. For readability, we omit the similar results for sessionization in these plots. To model the 0.9-quantile, we consider the cases that (1) the latency distribution L can be *observed* at runtime, for which we empirically observed it to be well approximated by normal distributions; (2) L is *not observable*, for which we model an upper bound of 0.9-quantile using Cantelli's inequality. In both workloads, the modeled latency has similar trends as the real values. When L is not observed, the modeled latency is an overestimate in all experiments, up to 70% of true latency, due to



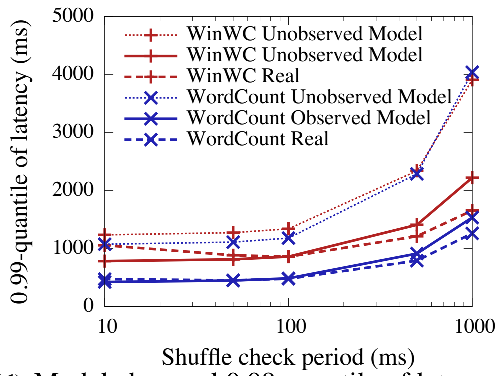
(a) Modeled vs real 0.9-quantile of latency with M tuned.



(b) Modeled vs real 0.9-quantile of latency with B_{ch} tuned.



(c) Modeled vs real 0.99-quantile of latency with M tuned.

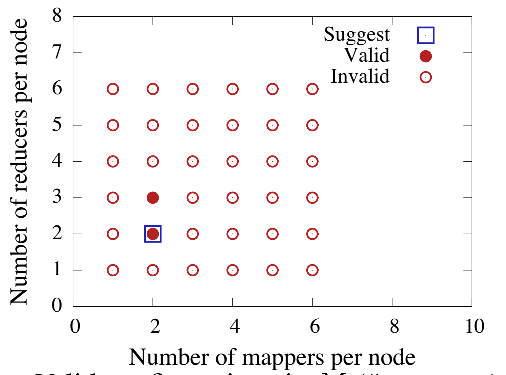


(d) Modeled vs real 0.99-quantile of latency with B_{ch} tuned.

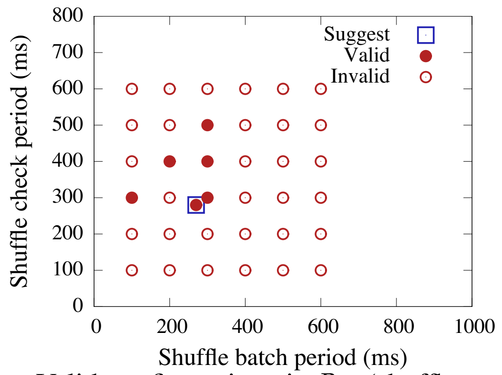
Figure 5.7. Accuracy of the model for quantiles of latency.

the use of a data-oblivious upper bound. When L is observable, which is expected to be the common case, the model accuracy is much improved: the relative error is below 20% in 21 out of the 22 experiments.

For the *0.99-quantile of latency*, Figure 5.7(c) and 5.7(d) show the results with varied M and B_{ch}, for word counting and windowed word counting. In both workloads, the modeled latency has similar trends as the real values. When L is not observed, the modeled latency is an overestimate in all experiments, up to 2.2 times of true latency, due to the use of a data-oblivious upper bound. When L is observable, which is expected to be the common case, the model accuracy is much improved: the relative error is below 20% in 19 out of the 22 experiments.



(a) Valid configurations in M (# mappers) and R (# reducers) under 1-sec constraint on avg. latency.

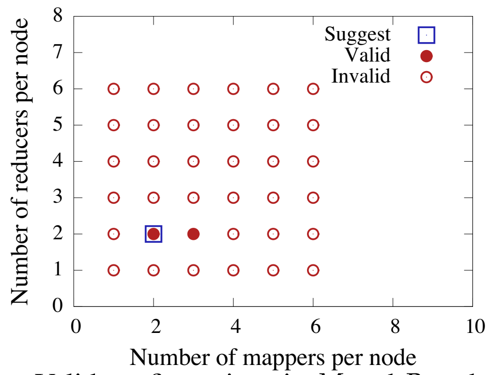


(b) Valid configurations in B_{sh} (shuffle period) and B_{ch} (shuffle check period) under 1-sec constraint on avg. latency.

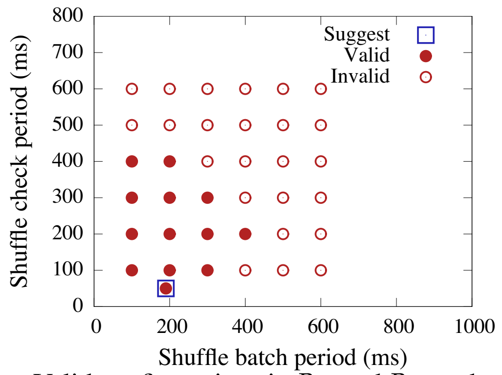
Figure 5.8. Validity of the model-driven configuration optimization under average latency.

Model Validity. We next validate the system configuration returned by our model under the word counting workload. We consider 1-second latency constraints on average latency, 0.9- and 0.99-quantiles of latency. We feed data at the maximum input rates according to our model, which are 1.2 million tweets/sec for the average latency, 1 million tweets/sec for the 0.9-quantile, and 0.81 million tweets/sec for the 0.99-quantile. We evaluate our system under the model-suggested configuration, as well as other configurations in the system parameter space. We run three times under each configuration, and consider a configuration valid if (1) the latency metric is below 1 sec, and (2) no input tuples are dropped, in all three runs. Figure 5.8(a) and 5.8(b) show the validity of configurations in the 2-dimensional space of M and R, and the space of B_{sh} and B_{ch}, respectively for average latency. We can see that there are only a few configurations valid at the input rate suggested by the model, marked by solid dots. The model-suggested configuration, marked by a square, is among those few valid configurations. Figure 5.9(a) and 5.9(b) show similar results for the 0.9-quantile of latency. Figure 5.9(c) and 5.9(d) show those for the 0.99-quantile of latency.

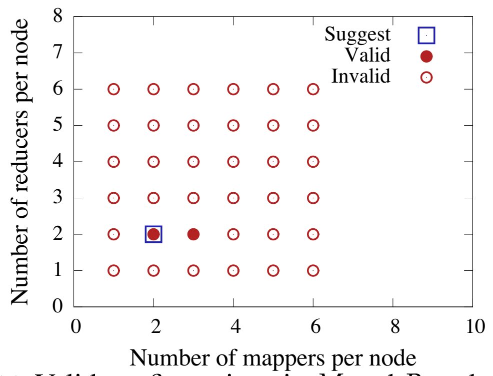
Choosing the Cluster Size. As stated earlier, our model-based approach can be used to decide the cluster size. Given different numbers of slave nodes, the maximum input rates suggested by our model for 1-second constraint on average latency, and 0.7-, 0.9- and



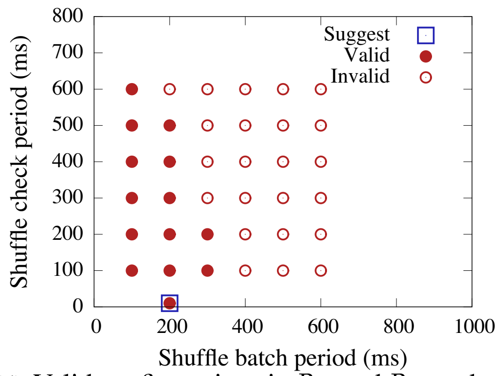
(a) Valid configurations in M and R under 1-sec constraint on 0.9-quantile of latency.



(b) Valid configurations in B_{sh} and B_{ch} under 1-sec constraint on 0.9-quantile of latency.



(c) Valid configurations in M and R under 1-sec constraint on 0.99-quantile of latency.

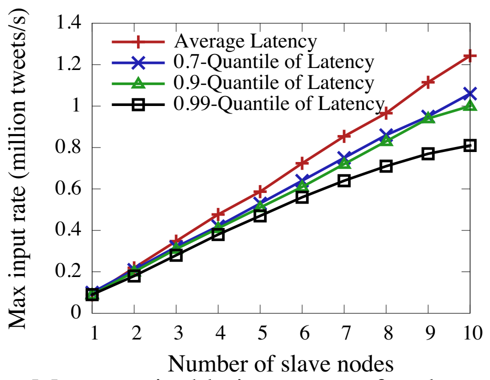


(d) Valid configurations in B_{sh} and B_{ch} under 1-sec constraint on 0.99-quantile of latency.

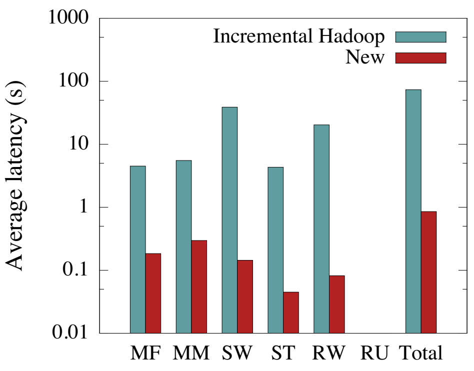
Figure 5.9. Validity of the model-driven configuration optimization under quantiles of latency.

0.99-quantiles of latency are shown in Figure 5.10(a). As such, a system can decide the cluster size based on the estimated input rate.

Comparison to Incremental Hadoop. Finally, we break down the average latency in our new system with the suggested configuration, and compare it with Incremental Hadoop, as shown in Figure 5.10(b). The breakdown is described in Section 5.1. We can see that the latency in each phase in our system is sub-second or even lower, due to the use of mini-batches for scheduling, with the appropriate batch sizes and number of processes per node chosen by our model.



(a) Max sustainable input rate of a cluster, with different latency metrics kept below 1 second.



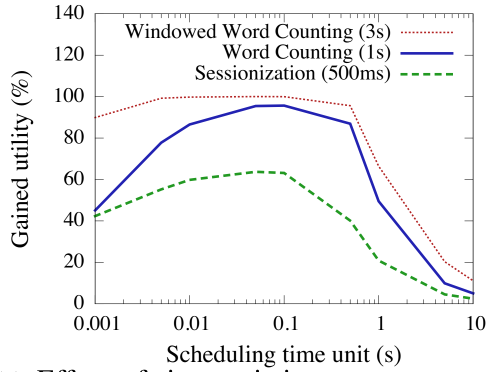
(b) Latency breakdown of Incremental Hadoop and our system configured for 1-sec avg. latency.

Figure 5.10. Results of our model-driven configuration optimization.

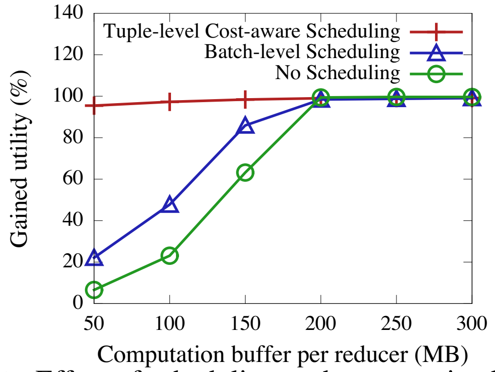
5.5.2 Latency-Aware Scheduling

We next evaluate the effectiveness of our runtime scheduling algorithms. The measurement is the percentage of gained utility (as defined in Section 5.4) over the maximum possible utility, i.e. the percentage of tuples that satisfy the latency constraint. An initial issue to resolve is how the size of time unit affects the effectiveness of our cost-aware scheduling algorithm. For all three workloads tested, we observe that the highest utility is gained when the time unit is about one order of magnitude smaller than the latency requirement (Figure 5.11(a)) – smaller time units incur significantly higher CPU cost in scheduling, and larger time units make the deadline for each tuple so inaccurate that the algorithm fails to prioritize the tuples effectively. Hence, in the following experiments we set the time unit of the cost-aware scheduling to 10% of the latency requirement.

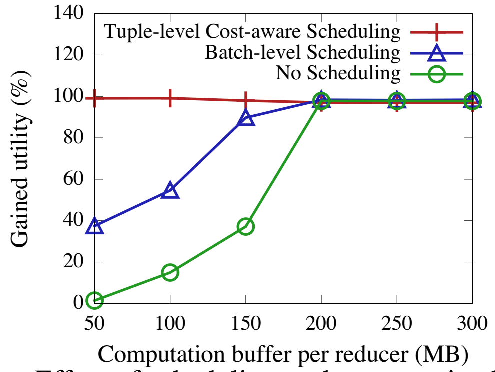
We now compare our scheduling algorithms in Section 5.4, by varying the memory size Υ in each reducer. Figure 5.11(b) shows gained utility under the 1-second latency constraint in the word counting workload with an input of 1.1 million tweets per second. Here, the minimum memory needed to hold all key-state pairs in a reducer is 200MB. As Υ reduces below 200MB, without scheduling the number of tuples that satisfy the latency constraint drops very fast. It is because now some key-state pairs have to be staged to the key-value



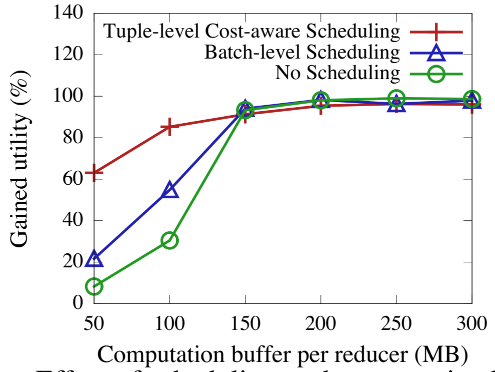
(a) Effect of time unit in our cost-aware scheduling.



(b) Effect of scheduling under constrained memory, in the word counting workload.



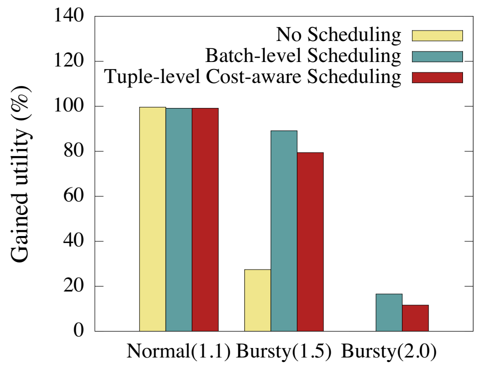
(c) Effect of scheduling under constrained memory, in windowed word counting.



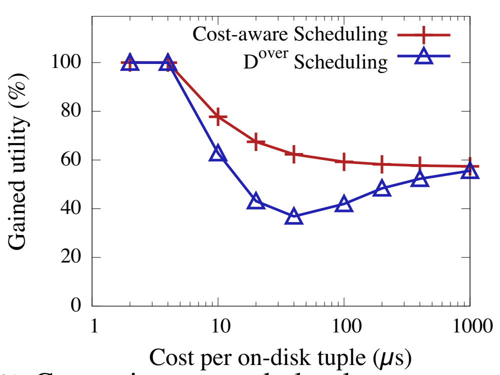
(d) Effect of scheduling under constrained memory, in the sessionization workload.

Figure 5.11. Results of runtime scheduling.

store on disk. Hence, the per-tuple processing cost increases, which in turn reduces the sustainable input rate to under 1.1 million per second. As tuples queue up in the system, when many of them arrive at the reducer, they have already missed the deadline. Processing them offers no utility and deprives other viable tuples of necessary resources, causing them to miss the deadline as well. Batch-level scheduling helps by dropping some batches when they arrive at the reducer. However, among those retained batches, those tuples whose key-states are on disk are still processed, postponing other viable cheap-to-process tuples until after the deadline. The cost-aware scheduling offers the best performance, without significant drop in utility. This is because the tuple-level, cost-aware scheduling gives higher priority to tuples whose key-states pairs are in memory. For skewed key distribution, it further keeps most



(a) Effect of scheduling under busty inputs.



(b) Comparing our tuple-level, cost-aware scheduling to the D^{over} algorithm.

Figure 5.12. Results of runtime scheduling.

hot keys in memory and thus most tuples are processed in memory. Figure 5.11(c) shows similar trends for the windowed word counting workload. Figure 5.11(d) shows the results for the sessionization workload. Different from the previous two workloads, the utility of cost-aware scheduling drops slightly, but is still much higher than the other two methods. The slight drop in utility is mainly due to less skewed frequencies of keys in this workload, and thus fewer tuples to process in memory.

We next consider bursty input. For word counting with sufficient memory under 1-second latency constraint, Figure 5.12(a) shows the results for the normal load of 1.1 million tweets/sec, the moderate overload of 1.5 million tweets/sec, and high overload of 2.0 million tweets/sec. In the moderate overload case, only a small fraction of tuples can meet the latency requirement without scheduling. With scheduling, by dropping non-viable tuples, enough system resources are saved to process the majority of tuples within latency constraint. In the high overload case, very few tuples can meet the latency constraint without scheduling (which is too low to be displayed in the figure). Scheduling helps increase the system utility, but many tuples are dropped since they have missed deadlines upon arrival at the reducer. Here scheduling works as a load-shedding mechanism, saving significant CPU cycles for non-viable tuples.

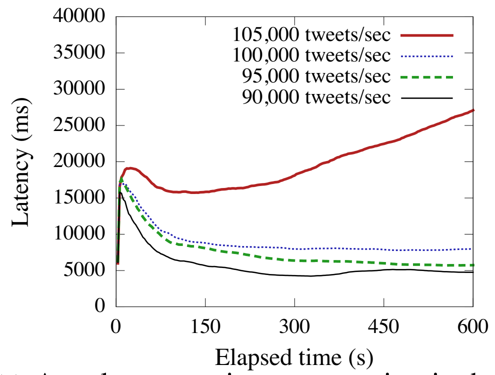
Last, we compare our cost-aware scheduling to D^{over} [52], a proven optimal algorithm in the worse case. For a thorough study with controlled per-tuple cost, we evaluate the algorithms with simulation based on the input trace to reducers collected in real workloads with constrained memory (50MB) per reducer. In the simulation, we vary the cost of an on-disk tuple while fix the cost of an in-memory tuple to a value measured in the real system. Figure 5.12(b) shows the results from the trace of the sessionization workload where the cost per in-memory tuple is $2\mu s$. We can see that when the cost per on-disk tuple is $10\mu s$ to $100\mu s$, our cost-aware scheduling outperforms D^{over} significantly due to the ability to prioritize in-memory tuples. In practice, SSDs are widely used for key-value stores, and a random I/O on a modern SSD takes tens of microseconds.[7] Thus, the cost per on-disk tuple may easily fall in the range between $10\mu s$ and $100\mu s$, in which case, our cost-aware scheduling shows superior performance over D^{over}. Similar observations are made in the word counting and windowed word counting workloads as well.

5.5.3 Comparison to Other Systems

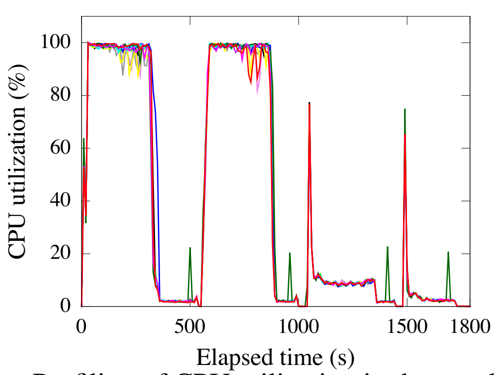
Storm. Storm [92] is an open-source distributed fault-tolerant stream system. A Storm cluster uses a master-slave architecture, and runs topologies that are similar to MapReduce jobs but can run continuously over streams. A topology is a directed acyclic graph of logical operators, which are called Spouts and Bolts. Spouts emit data as source streams of the system and Bolts consume input streams, do data processing, and possibly emit new streams to downstream operators. Storm supports shuffling between operators via various types of grouping, where field grouping is the same as the partitioning of map output in MapReduce.

We extended Storm 0.9.0 with two new pieces of code: (1) the mechanism for handling out-of-order data as our system, including the use of low watermarks to produce complete results (Section 5.2.2); (2) a distributor to control the input rate.We implemented the same

[7]For example, Samsung 850 PRO SSD can achieve 90K to 100K IOPS, which translates to 10 to $11\mu s$ per random I/O.



(a) Avg. latency as input rate varies, in the word counting workload.



(b) Profiling of CPU utilization in the word counting workload.

Figure 5.13. Evaluation of Storm on latency and throughput.

workloads as used in previous experiments. For word counting and windowed word counting, the topology is such that Spouts read tweets from the distributor and split them into words, and then send the words to Bolts to count the frequency of these words. For sessionization, the Spouts read clicks from the distributor, and extract and send user ids to Bolts to compute sessions. There are 20 Spouts and 20 Bolts running in parallel. For fair comparison, we make sure that configurations of Storm and our system are the same, including the same number of tasks per node and the same amount of memory for each task and each queue.

Figure 5.13(a) shows how the latency of Storm stabilizes over time, using the word counting workload. Here each line represents an input rate, R, to the 10-node cluster. When $R = 100{,}000$ tweets/sec or lower, the latency stabilizes, e.g., 8 second for the 100,000 input rate. However, for $R > 100{,}000$ tweets/sec, the latency increases continuously. In this case, the processing in Storm cannot keep pace with the input rate, so the tuples accumulate between the spouts and bolts, making the latency increase continuously. For windowed word counting and sessionization, similar trends are observed and the maximum input rates with stable latency are 95,000 tweet/sec and 1 million clicks/sec, respectively.

We examined the system profiling results and found the CPU utilization to be near 100% in the case when the latency does not stabilize. In Figure 5.13(b), we record the CPU utilization in four scenarios. We run (1) the word counting workload from 0 to 300 seconds;

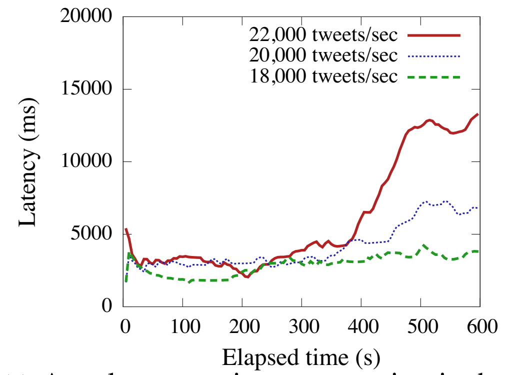
(2) from 550-850 seconds, the revised task where Bolts do nothing but simply drop the received data, with the CPU utilization still close to 100%; (3) from 1050 to 1350 seconds, the revised task with no Bolts in the topology, leading to much lower CPU utilization; (4) from 1500 to 1800 seconds, the task where Spouts do not process data, again with low CPU utilization. These results show that the shuffling overhead between Spouts and Bolts is the bottleneck of the system. Since Storm handles shuffling and transmission for every tuple, CPU becomes the bottleneck.[8]

Spark Streaming. Spark [98] is an open-source, fault-tolerant parallel processing system tailored for in-memory computation. In Spark, data is stored in a distributed in-memory data abstraction, called Resilient Distributed Datasets (RDDs). Spark transforms an RDD to another RDD by user-defined coarse-grained transformations, including `map` and `reduce`, and other transformations such as `sample`, `distinct`, `union` and `intersection`. An analytical job consists of one or multiple transformations. Spark provides fault-tolerance by tracking the lineage of each RDD, and re-computing any lost partition of an RDD. **Spark Streaming [99]** is a functionality integrated in Spark that enables parallel stream processing. Spark Streaming periodically creates a mini-batch containing the streaming input data, and transforms stream processing into a sequence Spark batch jobs over input mini-batches.

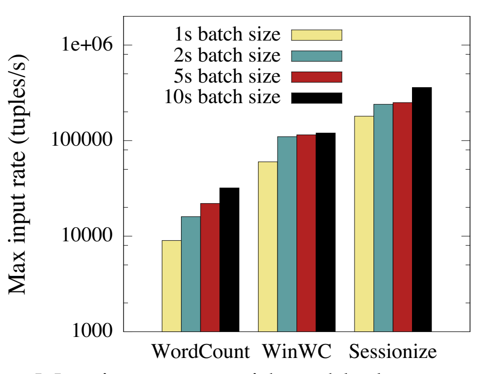
We extended Spark 1.0[9] with a custom *Receiver* to connect to our data distributor that controls the input rate in experiments. We implemented the same workloads as in previous experiments. The word counting workload uses a $flatMap$ transformation that splits tweets into words, and $updateStateByKey$ that computes the frequency of each word. The windowed word counting workload uses the same $flatMap$ to split words and $reduceByKeyAndWindow$ to compute word frequencies in each time window. The

[8]We also downloaded a new version of Storm, called Trident, which uses a new abstraction to process streams in batches. However, we observe the performance to be worse than Storm due to various added overheads.

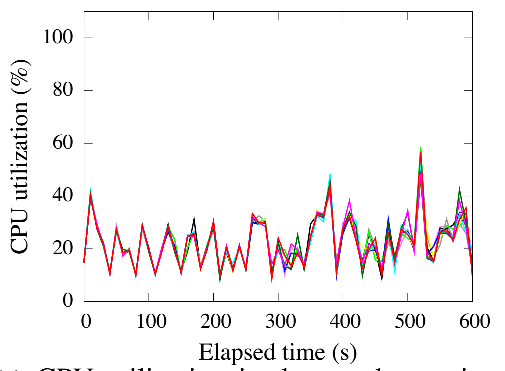
[9]A patch that fixed a bug for windowed streaming jobs is applied: `https://github.com/apache/spark/pull/961`



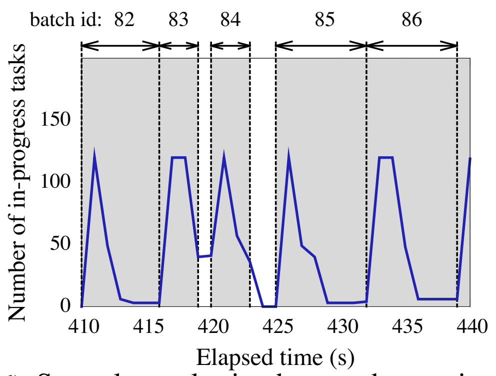
(a) Avg. latency as input rate varies, in the word counting workload.



(b) Max input rate with stable latency as batch size varies.



(c) CPU utilization in the word counting workload with overload.



(d) Straggler tasks in the word counting workload.

Figure 5.14. Evaluation of Spark Streaming on latency and throughput.

sessionization workload uses *map* to extract user IDs and *updateStateByKey* to compute sessions of each user. For fair comparison, we set the number of concurrent tasks per node to 4, which equals the total number of mappers and reducers per node in our system.[10] We also make sure the same amount of memory is used per node.

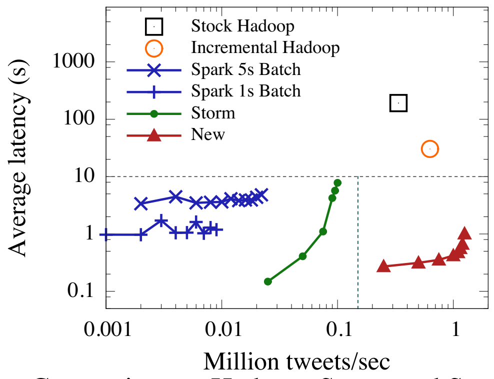
Figure 5.14(a) shows how the latency in Spark Streaming stabilizes over time with 5sec batch size in the word counting workload. We made similar observations in windowed word counting and sessionization. The maximum input rates with stable latency in the three workloads are 20,000 tweets/sec, 115,000 tweets/sec and 250,000 clicks/sec, respectively.

[10]We set NUM_CORES to 6, where 2 "cores" are taken to receive input data and 4 "cores" are available for computation on each machine.

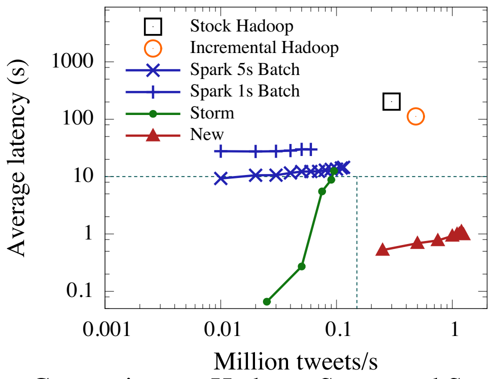
Figure 5.14(b) shows the maximum sustained input rate under different batch sizes. In all the three workloads, Spark Streaming can sustain a higher input rate when the batch size is larger due to smaller overhead to start Spark batch jobs.

To understand the bottleneck of Spark Streaming, we examined the system profiling results and found CPU, network and disk are all under-utilized. For example, Figure 5.14(c) shows the CPU utilization in the word counting workload with 5sec batch when input rate is 22,000 tweets/sec, which is higher than the max input rate with stable latency. The average CPU utilization is 21.6%.[11] The combination of the following two reasons prevents Spark Streaming from fully utilizing system resource and supporting higher input rate. (1) *Straggler tasks for a batch*: The computation of a batch consists of a large number of distributed tasks, and the completion of processing a batch is determined by the last finished task. Figure 5.14(d) shows the number of in-progress tasks in the system during a 30-second period in the word counting workload, with the corresponding time interval of processing each batch marked at the top. For batch 82, only a few straggler tasks remain running in the second half of the time interval, and delay the completion of the processing of the batch. Similar observations can be made for batch 85 and batch 86. The major cause of the stragglers in this example is the unstable time taken by data shuffling. Other reasons may also result in stragglers, such as data or computation skew, JVM garbage collection and failure. (2) *Blocking between batches*: Spark Streaming only starts the processing of a batch after all the tasks of the previous batch complete. As shown in Figure 5.14(d), where the batch size is 5sec, the processing of batch 82, 83, 84, 85 and 86 should start at 410s, 415s, 420s, 425s and 430s, respectively. However, it can be seen that batch 82 delays the start of batch 83, and batch 85 delays batch 86. Due the blocking between batches, the system resource cannot be effectively utilized during the occurrence of any straggler. Thus,

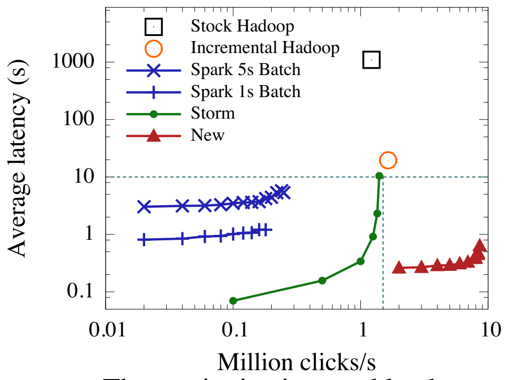
[11]Despite various approaches to improving the resource utilization, including the exploration of various combinations of relevant system parameters and the optimization of user code, no obvious improvement is observed. Similar observations are made in the user community of Spark Streaming.



(a) Comparison to Hadoop, Storm and Spark (the word counting workload).



(b) Comparison to Hadoop, Storm and Spark (the windowed counting workload).



(c) The sessionization workload.

Figure 5.15. Comparing Hadoop, INC-hash, Spark Streaming, Storm and our system on latency and throughput.

the combination of stragglers and the blocking fashion of processing batches becomes the bottleneck of system throughput.

Put All Systems Together. Here, we summarize the performance of stock Hadoop, Incremental Hadoop, Storm [92], Spark Streaming [99] and our new platform. Figure 5.15(a) shows the results of word counting in the two-dimensional space of latency and throughput achieved. We roughly partition the space into three regions: (1) Stock Hadoop and Incremental Hadoop read data from the distributed file system, and hence have a fixed input rate of around 300,000 and 600,000 tweets/sec, respectively. However, both have high tuple latency, in the range of tens to hundreds of seconds, hence not suitable for low-latency tasks. (2) Storm achieves stable latency of less than 10 seconds when the input rate is at most

100,000 tweets/sec, belonging to the low input rate, low latency region. (3) Spark Streaming achieves stable latency of less than 10 seconds when the input rate is at most 9,000 tweets/sec for 1sec batch size, or at most 20,000 tweets/sec for 5sec batch size, which puts it also in the the low input rate, low latency region. (4) Our new platform, when configured for 1-second average latency and with scheduling turned off, is able to fully process every input tuple with a mean latency under 1sec for input rates up to 1.2 million tweets/sec, putting it in the high input rate, low latency region. With the scheduler turned on, our system allows more tuples to satisfy the latency constraint, providing better utility. Similar results for the sessionization workload are shown in Figure 5.15(c).

For the windowed word counting workload, Spark Streaming stays in the low input rate region but gets close to the high latency region, as shown in Figure 5.15(b). This is because Spark Streaming does not perform incremental processing for a window. Instead, it performs all computation of a window after the window ends, and thus delivers output of the window with high latency.

5.5.4 Summary

Our new platform can reduce the average latency from 10's of seconds in Incremental Hadoop to sub-second, with 2x-5x increase in throughput. It is able to outperform Storm by 7x-28x in latency and 8-13x in throughput, and outperform Spark Streaming by 4x-27x in latency and 10x-56x in throughput.

5.6 Related Work

In this section, we discuss the literature relevant to our latency models and runtime scheduling techniques.

Queueing Theory. Queueing Theory [58, 87, 50, 46, 51, 74, 6] studies the statistical properties of customers in a service system, such as mean waiting time and distribution of the number of customers in queues. For single-server models, M/M/1 has been thoroughly studied, where the times between successive customer arrivals are iid exponential random variables, and the service times of customers are also iid exponential random variables. Analytical solutions of mean waiting time and distribution of the number of customers in the queue are known [6]. Jansson [46] showed the analytical solutions for D/M/1, where the service times are still exponentially distributed, but arrivals occur at a constant rate. Lindley [58] and Smith [87] studied a more general single-server model G/G/1, where interarrival times and service times have general distributions. However, the solution of waiting time is difficult to evaluate. Kingman [50] showed an upper bound of mean waiting time for G/G/1. The upper bound relies only on the means and variances of the interarrival time and service time distributions, and is asymptotically sharp in heavy traffic [51]. Ott [74] improved the bound of Kingman for light traffic under D/G/1, a sub-class of G/G/1 where arrivals occur at a constant rate. Higher moments of waiting time in G/G/1 have also been studied [6].

Online Scheduling. The problem of maximizing total utility in this context is close to the problems of online scheduling with deadline in real-time operating systems [26, 67, 60, 81, 83, 9, 84, 52, 21], where computation tasks are scheduled to maximize utility. Scheduling in underloaded systems has been well studied. Optimal online algorithms have been proposed for uniprocessor systems [26, 67]. However, these algorithms do not provide performance guarantees for overloaded systems and have shown poor performance in experiments [60]. Scheduling algorithms tailored for overloaded situations usually employ an additional test module [81, 83, 9, 84, 52]. Given a newly released tasks, the test module checks whether the new task will cause overload of the existing tasks in the current schedule. The module decides to accept the new task if no overload is caused. Otherwise, it rejects the new task or abandons some existing tasks. Heuristic-based methods have been proposed for uniprocessor [9, 84]. However, they do not provide guarantee on the worst case competitive ratio. Baruah et al. [10] provided an upper bound of competitive ratio for uniprocessor $1/(1+\sqrt{k})^2$, where k is the importance ratio. Koren et al. [52] designed an online scheduling algorithm,

D^{over}, for uniprocessor. D^{over} is optimal in underloaded environment and achieves the competitive ratio of $1/(1+\sqrt{k})^2$ in the worst case with presence of overload. However, D^{over} does not take the value density of tasks into consideration, and thus may suffer from low competitive ratio in practice. The two major differences in our problem are: (1) a tuple represents a task so the number of tasks to schedule is much larger, and (2) processing a tuple is relatively cheap compared to processing a complex task. Therefore, the existing scheduling techniques are not a good match of our problem due to the high time and space complexity of running the scheduling algorithm for a large number of cheap tuples.

Sparrow [75] is a scalable scheduling system that assigns high-volume short tasks to a set of machines. Sparrow achieves high scalability and fault-tolerance by using distributed and stateless schedulers without centralized state. Sparrow also supports task placement constraints, such as data locality. In our work, Sparrow can be used to distribute mini-batches to mappers. However, the objective of Sparrow is to minimize response time of tasks in a parallel environment by load balancing, different from our focus on constraints of per-tuple latency experienced in a sequence of operators.

5.7 Summary

Towards building a unified processing framework for big and fast data, we identified the causes of high latency in today's systems that support data parallelism and incremental processing. To support streaming processing with latency constraints, we proposed an extended architecture with mini-batches as granularity for computation and shuffling, and augmented it with new modeling and scheduling techniques to meet user-specified latency requirements while maximizing throughput. Results using real-world workloads show that our techniques, all implemented in Incremental Hadoop, can reduce average latency from 10's of seconds to sub-second, with a 2x-5x increase in throughput. Our scheduling techniques further increase the number of tuples that actually meet the latency constraint. Our new platform is able to outperform two state-of-the-art distributed stream systems, Storm and Sparking Streaming, by 1-2 orders of magnitude when considering both latency and throughput.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

This chapter summarizes the contributions made in this thesis work and states some research directions for future work.

6.1 Thesis Summary

Recently, there have been increasing needs of processing not only "big data" but also "fast data", which refers to high-speed real-time data streams, such as Twitter feeds, search query streams, click streams, impressions, and system logs. The needs call for a *general, unified* system to support analytics with different latency requirements. In order to build such a system, I propose to build on existing solutions on *data parallelism* and extend them with two new features: (1) *incremental processing* and (2) *stream processing with latency constraints*. I start with Hadoop, the most popular open-source MapReduce implementation, which provides proven scalability based on data parallelism. This thesis answers the following questions: (1) Is Hadoop able to support incremental processing? (2) What are the necessary architecture changes in order to support incremental processing? (3) What are the additional design features required to support streaming analytical queries with stringent latency requirements? To address these questions, the thesis includes the following three parts: Hadoop benchmarking and optimization, incremental processing, and stream processing with latency constraints. Each part is summarized as follows.

6.1.1 Hadoop Benchmarking and Optimization

Incremental processing means that computation is performed as soon as the relevant data becomes available. Incremental processing enables tuples to move quickly through a pipeline of operators, and thus is essential to returning query answers with low latency. The first part of the thesis answers the following question: “Is the existing MapReduce implementation, after tuning its parameters for optimization, able to support incremental processing?”

I first conduct a thorough benchmarking study, evaluating existing MapReduce platforms including Hadoop and MapReduce Online. The results reveal that the main mechanism for parallel processing used in these systems, based on a sort-merge technique, is subject to significant I/O bottlenecks as well as blocking: Next, building on these benchmarking results, I perform an in-depth analysis of Hadoop, using a theoretically sound analytical model to explain the empirical results. The key results are two-fold: (1) It is shown that the analytical model can be used to choose appropriate values of Hadoop parameters, thereby reducing I/O and startup costs. (2) Despite a range of optimizations, I/O bottlenecks as well as blocking persist, and the reduce progress falls significantly behind the map progress, hence violating the requirements of efficient incremental processing. Both theoretical and empirical analyses show that the sort-merge implementation, used to support data parallel processing, poses a fundamental barrier to incremental processing.

6.1.2 Incremental Processing

Based on the insight that the sort-merge implementation in the original MapReduce model poses a fundamental barrier to incremental processing, the next question to answer is: “What are the necessary architecture changes in MapReduce in order to support incremental processing?” In the second part of the thesis, I propose a new data analysis platform based on MapReduce that is geared for incremental processing.

I made two key architecture changes to Hadoop. The first mechanism replaces the sort-merge implementation in Hadoop with purely hash-based techniques. These hash techniques can provide fast in-memory processing of the reduce function when the memory is sufficient. The second mechanism further brings the benefits of fast in-memory processing to workloads that require a large key-state space that far exceeds available memory. I propose techniques to dynamically recognize popular keys and then update their states using a full in-memory processing path, both saving I/Os and enabling early answers for these keys. Less popular keys trigger I/Os to stage data to disk but have limited impact on the overall efficiency.

Experiments on a range of workloads in click stream analysis and web document analysis show the following main results: (1) Given sufficient memory, my hash techniques enable fast in-memory processing of the reduce function. (2) For challenging workloads that require a large key-state space, my dynamic hashing mechanism allows the reduce progress to keep up with the map progress with up to 3 orders of magnitude reduction of internal disk spills. (3) Further trade-offs exist between my hash-based techniques under different workload types, data localities, and memory sizes, with dynamic hashing working the best under constrained memory and most workloads.

6.1.3 Stream Processing with Latency Constraints

The revised MapReduce platform offers data parallelism and incremental processing. The third part of the thesis answers the following questions: "Which additional design features are needed to support streaming analytical queries with stringent latency requirements?"

First, I conduct a benchmark study in order to understand the sources of latency in a system supporting data parallelism and incremental processing. The results call for *job-specific resource planning* to select the appropriate parameter settings and *latency-aware scheduling* to determine which tuples to process and in what order to process them in order to keep latency bounded. For resource planning, I propose a model-driven approach to automatically determining the resource allocation plan for each job. I formulate the per-job resource planning problem as a constrained optimization problem, where the constraint depends on the modeling of latency. Hence, I further develop various latency models for a collection of widely-used latency metrics, including per-tuple latency, per-window latency, and any quantiles associated with these latency distributions. For latency-aware scheduling, I propose runtime scheduling algorithms to maximize the number of results that meet the latency requirement, i.e., the *total utility*. I propose two runtime scheduling algorithms, at batch-level and tuple-level, respectively, which consider both costs and deadlines of data processing. In particular, my tuple-level scheduling algorithm has provable results on the quality of runtime schedules and efficiency of the scheduling algorithm.

Evaluation using real-world workloads shows: (1) My model-driven approach to resource planning can reduce the average latency from 10's of seconds in Incremental Hadoop to sub-second, with 2x-5x increase in throughput. (2) My latency-aware scheduling can dramatically improve the number of tuples meeting the latency constraint, especially under constrained memory. (3) My system offers 1-2 orders of magnitude improvements over Twitter Storm and Spark Streaming, two state-of-the-art distributed stream systems, when considering both latency and throughput.

6.1.4 Resulting System

I have implemented all of the proposed techniques in Hadoop. The implementation consists of a library of (1) core algorithms for hash-based incremental processing, latency modeling and runtime scheduling, (2) efficient binary in-memory data structures such as hash tables, queues and buffers, and (3) implementations that manage local files, key-value stores, network transmission and paging. Based on this library, I mainly modified the `MapTask` and `ReduceTask` modules in Hadoop to revise the processing of `map` and `reduce` functions. The implementation includes about 28,000 lines of Java code. The resulting system can run (1) in the original **batch mode**, fully supporting existing analytic tasks in enterprise businesses, or (2) in the new **streaming mode** where user-specified latency constraints are handled automatically by the system and used to guide resource allocation and scheduling, with tremendous performance benefits over existing parallel stream systems. The first version of the system built on Hadoop 0.20.1 has been released as SCALLA 0.1 [62].

6.2 Future Work

This section discusses some future research directions that have emerged from this thesis.

- ▶ **Automatic Mode Selection.** This thesis has taken a step towards building a general, unified system for data analytics under different latency requirements. The resulting system from this thesis supports both the batch mode and the streaming mode. Only one system needs to be maintained for processing "big data" and "fast data". However, the choice of the mode still needs to be made manually. It is desirable to decide the mode automatically based on the latency constraint, workload characteristics, and available resource.
- ▶ **Resource Planning for Multiple Jobs.** The resource planning proposed in this thesis can support concurrent jobs by solving a constraint optimization problem for each job, and allocating each job on a separate set of machines. However, solving the resource planning problem for multiple jobs in a holistic manner opens the opportunity to allocate jobs with different characteristics, such as a CPU intensive job and a network intensive job, on the same set of nodes, and may further reduce the number of required machines by more fully utilizing resources, while satisfy the same latency and throughput requirements.
- ▶ **Runtime Scheduling under Other Utility Models.** This thesis proposes runtime scheduling in order to maximize the gained utility. The adopted utility model is a step function, which is suitable for workloads with hard latency constraints. However, the latency constraints in some workloads are soft, which requires more complex utility models. So, runtime scheduling with more general utility models could be a potential direction of future work.

BIBLIOGRAPHY

[1] Abadi, Daniel J, Ahmad, Yanif, Balazinska, Magdalena, Cetintemel, Ugur, Cherniack, Mitch, Hwang, Jeong-Hyon, Lindner, Wolfgang, Maskey, Anurag S, Rasin, Alexander, Ryvkina, Esther, Tatbul, Nesime, Xing, Ying, and Zdonik, Stan. The Design of the Borealis Stream Processing Engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)* (Asilomar, CA, January 2005).

[2] Abouzeid, Azza, Bajda-Pawlikowski, Kamil, Abadi, Daniel J., Rasin, Alexander, and Silberschatz, Avi. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. *PVLDB 2*, 1 (2009), 922–933.

[3] Afrati, Foto N., Borkar, Vinayak R., Carey, Michael J., Polyzotis, Neoklis, and Ullman, Jeffrey D. Map-reduce extensions and recursive queries. In *EDBT* (2011), pp. 1–8.

[4] Akidau, Tyler, Balikov, Alex, Bekiroglu, Kaya, Chernyak, Slava, Haberman, Josh, Lax, Reuven, McVeety, Sam, Mills, Daniel, Nordstrom, Paul, and Whittle, Sam. Millwheel: Fault-tolerant stream processing at internet scale. In *Very Large Data Bases* (2013), pp. 734–746.

[5] Ananthanarayanan, Rajagopal, Basker, Venkatesh, Das, Sumit, Gupta, Ashish, Jiang, Haifeng, Qiu, Tianhao, Reznichenko, Alexey, Ryabkov, Deomid, Singh, Manpreet, and Venkataraman, Shivakumar. Photon: fault-tolerant and scalable joining of continuous data streams. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2013), SIGMOD '13, ACM, pp. 577–588.

[6] Asmussen, Søren. *Applied probability and queues; 2nd ed.* Stochastic Modelling and Applied Probability. Springer, New York, 2003.

[7] Babcock, B., Datar, M., and Motwani, R. Load shedding for aggregation queries over data streams. In *Data Engineering, 2004. Proceedings. 20th International Conference on* (March 2004), pp. 350–361.

[8] Babu, Shivnath. Towards automatic optimization of mapreduce programs. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)* (New York, NY, USA, 2010), ACM, pp. 137–142.

[9] Baker, T., and Shaw, Alan. The cyclic executive model and ada. *Real-Time Systems 1* (1989), 7–25.

[10] Baruah, S., Koren, G., Mishra, B., Raghunathan, A., Rosier, L., and Shasha, D. On-line scheduling in the presence of overload. In *Foundations of Computer Science, 1991. Proceedings., 32nd Annual Symposium on* (1991), pp. 100 –110.

[11] Berinde, Radu, Cormode, Graham, Indyk, Piotr, and Strauss, Martin J. Space-optimal heavy hitters with strong error bounds. In *PODS* (2009), pp. 157–166.

[12] Borkar, Vinayak R., Carey, Michael J., Grover, Raman, Onose, Nicola, and Vernica, Rares. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE* (2011), pp. 1151–1162.

[13] Borthakur, Dhruba, Gray, Jonathan, Sarma, Joydeep Sen, Muthukkaruppan, Kannan, Spiegelberg, Nicolas, Kuang, Hairong, Ranganathan, Karthik, Molkov, Dmytro, Menon, Aravind, Rash, Samuel, Schmidt, Rodrigo, and Aiyer, Amitanand. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2011), SIGMOD '11, ACM, pp. 1071–1080.

[14] Brito, A., Martin, A., Knauth, T., Creutz, S., Becker, D., Weigert, S., and Fetzer, C. Scalable and low-latency data processing with stream mapreduce. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on* (Nov 2011), pp. 48–58.

[15] Carney, Don, Çetintemel, Uğur, Rasin, Alex, Zdonik, Stan, Cherniack, Mitch, and Stonebraker, Mike. Operator scheduling in a data stream manager. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29* (2003), VLDB '03, VLDB Endowment, pp. 838–849.

[16] Castro Fernandez, Raul, Migliavacca, Matteo, Kalyvianaki, Evangelia, and Pietzuch, Peter. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2013), SIGMOD '13, ACM, pp. 725–736.

[17] Chaiken, Ronnie, Jenkins, Bob, Larson, Per-Åke, Ramsey, Bill, Shakib, Darren, Weaver, Simon, and Zhou, Jingren. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow. 1*, 2 (2008), 1265–1276.

[18] Chandrasekaran, Sirish, Cooper, Owen, Deshpande, Amol, Franklin, Michael J., Hellerstein, Joseph M., Hong, Wei, Krishnamurthy, Sailesh, Madden, Samuel, Raman, Vijayshankar, Reiss, Frederick, and Shah, Mehul A. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR* (2003).

[19] Chang, Fay, Dean, Jeffrey, Ghemawat, Sanjay, Hsieh, Wilson C., Wallach, Deborah A., Burrows, Mike, Chandra, Tushar, Fikes, Andrew, and Gruber, Robert E. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 205–218.

[20] Charikar, Moses, Chen, Kevin, and Farach-Colton, Martin. Finding frequent items in data streams. *Theor. Comput. Sci. 312*, 1 (2004), 3–15.

[21] Chin, Francis Y.L., and Fung, Stanley P.Y. Improved competitive algorithms for online scheduling with partial job values. *Theoretical Computer Science 325*, 3 (2004), 467 – 478. Selected Papers from {COCOON} 2003.

[22] Condie, Tyson, Conway, Neil, Alvaro, Peter, Hellerstein, Joseph M., Elmeleegy, Khaled, and Sears, Russell. Mapreduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation* (Berkeley, CA, USA, 2010), NSDI'10, USENIX Association, pp. 21–21.

[23] Cormode, Graham, and Muthukrishnan, S. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms 55*, 1 (2005), 58–75.

[24] David, H.A., and Nagaraja, H.N. *Order Statistics*. Wiley, 2004.

[25] Dean, Jeffrey, and Ghemawat, Sanjay. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association, pp. 10–10.

[26] Dertouzos, M. L. Control robotics: The procedural control of physical processes. In *IFIF Congress* (1974), pp. 807–813.

[27] DeWitt, David, and Gray, Jim. Parallel database systems: the future of high performance database systems. *Commun. ACM 35*, 6 (1992), 85–98.

[28] DeWitt, David J., Gerber, Robert H., Graefe, Goetz, Heytens, Michael L., Kumar, Krishna B., and Muralikrishna, M. Gamma - a high performance dataflow database machine. In *VLDB* (1986), pp. 228–237.

[29] DeWitt, David J., Ghandeharizadeh, Shahram, Schneider, Donovan A., Bricker, Allan, Hsiao, Hui-I, and Rasmussen, Rick. The gamma database machine project. *IEEE Trans. Knowl. Data Eng. 2*, 1 (1990), 44–62.

[30] Ferguson, Andrew D., Bodik, Peter, Kandula, Srikanth, Boutin, Eric, and Fonseca, Rodrigo. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 99–112.

[31] Fiat, Amos, Karp, Richard M., Luby, Michael, McGeoch, Lyle A., Sleator, Daniel D., and Young, Neal E. Competitive paging algorithms. *J. Algorithms 12*, 4 (1991), 685–699.

[32] Friedman, Eric, Pawlowski, Peter, and Cieslewicz, John. Sql/mapreduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *PVLDB 2*, 2 (2009), 1402–1413.

[33] Ganguly, Sumit, and Majumder, Anirban. Cr-precis: A deterministic summary structure for update data streams. In *ESCAPE* (2007), pp. 48–59.

[34] Gates, Alan, Natkovich, Olga, Chopra, Shubham, Kamath, Pradeep, Narayanam, Shravan, Olston, Christopher, Reed, Benjamin, Srinivasan, Santhosh, and Srivastava, Utkarsh. Building a highlevel dataflow system on top of mapreduce: The pig experience. *PVLDB 2*, 2 (2009), 1414–1425.

[35] Gulisano, Vincenzo, Jimenez-Peris, Ricardo, Patino-Martinez, Marta, Soriente, Claudio, and Valduriez, Patrick. Streamcloud: An elastic and scalable data streaming system. *IEEE Trans. Parallel Distrib. Syst. 23*, 12 (Dec. 2012), 2351–2365.

[36] Hadoop online prototype (hop). `http://code.google.com/p/hop/`.

[37] Hadoop: Reliable, scalable, distributed computing. `http://hadoop.apache.org/`.

[38] Hbase: A distributed, scalable, big data store. `http://hbase.apache.org/`.

[39] Hellerstein, Joseph M., and Naughton, Jeffrey F. Query execution techniques for caching expensive methods. In *SIGMOD Conference* (1996), pp. 423–434.

[40] Herodotou, Herodotos. Hadoop performance models. *Technical Report CS-2011-05, Duke Computer Science* (2011).

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

[42] Herodotou, Herodotos, Dong, Fei, and Babu, Shivnath. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing* (New York, NY, USA, 2011), SOCC '11, ACM, pp. 18:1–18:14.

[43] Hive: Querying and managing large datasets residing in distributed storage. `http://hive.apache.org/`.

[44] Ibm: Big data at the speed of business. `http://www-01.ibm.com/software/data/bigdata/what-is-big-data.html`.

[45] Jalaparti, Virajith, Ballani, Hitesh, Costa, Paolo, Karagiannis, Thomas, and Rowstron, Ant. Bridging the tenant-provider gap in cloud services. In *Proceedings of the Third ACM Symposium on Cloud Computing* (New York, NY, USA, 2012), SoCC '12, ACM, pp. 10:1–10:14.

[46] Jansson, Birger. Choosing a good appointment system-a study of queues of the type (d, m, 1). *Operations Research 14*, 2 (1966), pp. 292–312.

[47] Jiang, Dawei, Ooi, Beng Chin, Shi, Lei, and Wu, Sai. The performance of mapreduce: an in-depth study. In *VLDB* (2010).

[48] Kane, Daniel M., Nelson, Jelani, and Woodruff, David P. An optimal algorithm for the distinct elements problem. In *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems* (New York, NY, USA, 2010), PODS '10, ACM, pp. 41–52.

[49] Karloff, Howard, Suri, Siddharth, and Vassilvitskii, Sergei. A model of computation for mapreduce. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (Philadelphia, PA, USA, 2010), Society for Industrial and Applied Mathematics, pp. 938–948.

[50] Kingman, J. F. C. Some inequalities for the queue gi/g/1. *Biometrika 49*, 3/4 (1962), pp. 315–324.

[51] Kingman, J. F. C. Inequalities in the theory of queues. *Journal of the Royal Statistical Society. Series B (Methodological) 32*, 1 (1970), pp. 102–110.

[52] Koren, G., and Shasha, D. Dover; an optimal on-line scheduling algorithm for overloaded real-time systems. In *Real-Time Systems Symposium, 1992* (dec 1992), pp. 290 –299.

[53] Krishnamurthy, Sailesh, Franklin, Michael J., Davis, Jeffrey, Farina, Daniel, Golovko, Pasha, Li, Alan, and Thombre, Neil. Continuous analytics over discontinuous streams. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (New York, NY, USA, 2010), SIGMOD '10, ACM, pp. 1081–1092.

[54] Kumar, Vibhore, Andrade, Henrique, Gedik, Buğra, and Wu, Kun-Lung. Deduce: at the intersection of mapreduce and stream processing. In *Proceedings of the 13th International Conference on Extending Database Technology* (New York, NY, USA, 2010), EDBT '10, ACM, pp. 657–662.

[55] Lam, Wang, Liu, Lu, Prasad, Sts, Rajaraman, Anand, Vacheri, Zoheb, and Doan, AnHai. Muppet: Mapreduce-style processing of fast data. *Proc. VLDB Endow. 5*, 12 (Aug. 2012), 1814–1825.

[56] Lee, L. K., and Ting, H. F. A simpler and more efficient deterministic scheme for finding frequent items over sliding windows. In *Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)* (New York, NY, USA, 2006), ACM, pp. 290–297.

[57] Li, Jin, Tufte, Kristin, Shkapenyuk, Vladislav, Papadimos, Vassilis, Johnson, Theodore, and Maier, David. Out-of-order processing: a new architecture for high-performance stream systems. *Proc. VLDB Endow. 1*, 1 (Aug. 2008), 274–288.

[58] Lindley, D. V. The theory of queues with a single server. *Mathematical Proceedings of the Cambridge Philosophical Society 48* (4 1952), 277–289.

[59] Liu, Bin, Zhu, Yali, and Rundensteiner, Elke. Run-time operator state spilling for memory intensive long-running queries. In *Proceedings of the 2006 ACM SIGMOD*

International Conference on Management of Data (New York, NY, USA, 2006), SIGMOD '06, ACM, pp. 347–358.

[60] Locke, C. D. Best-effort decision making for real-time scheduling. *Ph.D. thesis, Carnegie-Mellon University, Pittsburgh, PA* (1986).

[61] Logothetis, Dionysios, Olston, Christopher, Reed, Benjamin, Webb, Kevin C., and Yocum, Ken. Stateful bulk processing for incremental analytics. In *SoCC '10: Proceedings of the 1st ACM symposium on Cloud computing* (New York, NY, USA, 2010), ACM, pp. 51–62.

[62] Mass scalla project. `http://scalla.cs.umass.edu/`.

[63] McGeoch, Lyle A., and Sleator, Daniel Dominic. A strongly competitive randomized paging algorithm. *Algorithmica 6*, 6 (1991), 816–825.

[64] Metwally, Ahmed, Agrawal, Divyakant, and El Abbadi, Amr. Efficient computation of frequent and top-k elements in data streams. In *Database Theory - ICDT 2005*, Thomas Eiter and Leonid Libkin, Eds., vol. 3363 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2005, pp. 398–412.

[65] Mishne, Gilad, Dalton, Jeff, Li, Zhenghua, Sharma, Aneesh, and Lin, Jimmy. Fast data in the era of big data: Twitter's real-time related query suggestion architecture. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2013), SIGMOD '13, ACM, pp. 1147–1158.

[66] Misra, Jayadev, and Gries, David. Finding repeated elements. *Sci. Comput. Program. 2*, 2 (1982), 143–152.

[67] Mok, A. K.-L. Fundamental design problems of distributed systems for the hard-real-time environments. *Ph.D. thesis, Massachusetts Institute of Technology, Boston, MA* (1983).

[68] Morton, Kristi, Balazinska, Magdalena, and Grossman, Dan. Paratimer: a progress indicator for mapreduce dags. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (New York, NY, USA, 2010), SIGMOD '10, ACM, pp. 507–518.

[69] Motwani, Rajeev, Widom, Jennifer, Arasu, Arvind, Babcock, Brian, Babu, Shivnath, Datar, Mayur, Manku, Gurmeet Singh, Olston, Chris, Rosenstein, Justin, and Varma, Rohit. Query processing, approximation, and resource management in a data stream management system. In *CIDR* (2003).

[70] Murray, Derek G., McSherry, Frank, Isaacs, Rebecca, Isard, Michael, Barham, Paul, and Abadi, Martín. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 439–455.

[71] Muthukrishnan, S. *Data Streams: Algorithms and Applications*. Now Publishers, 2006.

[72] Neumeyer, L., Robbins, B., Nair, A., and Kesari, A. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on* (2010), pp. 170–177.

[73] Olston, Christopher, Reed, Benjamin, Srivastava, Utkarsh, Kumar, Ravi, and Tomkins, Andrew. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference* (2008), pp. 1099–1110.

[74] Ott, Teunis J. Simple inequalities for the d/g/1 queue. *Operations Research 35*, 4 (1987), pp. 589–597.

[75] Ousterhout, Kay, Wendell, Patrick, Zaharia, Matei, and Stoica, Ion. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 69–84.

[76] Pavlo, Andrew, Paulson, Erik, Rasin, Alexander, Abadi, Daniel J., DeWitt, David J., Madden, Samuel, and Stonebraker, Michael. A comparison of approaches to large-scale data analysis. In *SIGMOD Conference* (2009), pp. 165–178.

[77] Peng, Daniel, and Dabek, Frank. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation* (2010).

[78] Pig: A platform for analyzing large data sets that consists of a high-level language. `http://hive.apache.org/`.

[79] Qian, Zhengping, He, Yong, Su, Chunzhi, Wu, Zhuojie, Zhu, Hongyu, Zhang, Taizhi, Zhou, Lidong, Yu, Yuan, and Zhang, Zheng. Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 1–14.

[80] Ramakrishnan, Raghu, and Gehrke, Johannes. *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[81] Ramamritham, K., and Stankovic, J.A. Dynamic task scheduling in hard real-time distributed systems. *Software, IEEE 1*, 3 (July 1984), 65 –75.

[82] Schneider, S., Andrade, H., Gedik, B., Biem, A., and Wu, Kun-Lung. Elastic scaling of data parallel operators in stream processing. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on* (May 2009), pp. 1–12.

[83] Schwan, K., and Zhou, H. Dynamic scheduling of hard real-time tasks and real-time threads. *IEEE Transactions on Software Engineering 18*, 8 (Aug 1992), 736 –748.

[84] Sha, Lui, Lehoczky, John P., and Rajkumar, Ragunathan. Solutions for some practical problems in prioritized preemptive scheduling. In *IEEE Real-Time Systems Symposium* (1986), IEEE Computer Society, pp. 181–191.

[85] Shapiro, Leonard D. Join processing in database systems with large main memories. *ACM Trans. Database Syst. 11*, 3 (1986), 239–264.

[86] Sleator, Daniel Dominic, and Tarjan, Robert Endre. Amortized efficiency of list update and paging rules. *Commun. ACM 28*, 2 (1985), 202–208.

[87] Smith, Walter L. On the distribution of queueing times. *Mathematical Proceedings of the Cambridge Philosophical Society null* (7 1953), 449–461.

[88] Sources and types of big data. `http://whatsthebigdata.com/2013/11/10/sources-and-types-of-big-data-infographic/`.

[89] Tatbul, Nesime, Çetintemel, Uğur, and Zdonik, Stan. Staying fit: Efficient load shedding techniques for distributed stream processing. In *Proceedings of the 33rd International Conference on Very Large Data Bases* (2007), VLDB '07, VLDB Endowment, pp. 159–170.

[90] Tatbul, Nesime, and Zdonik, Stan. Window-aware load shedding for aggregation queries over data streams. In *Proceedings of the 32Nd International Conference on Very Large Data Bases* (2006), VLDB '06, VLDB Endowment, pp. 799–810.

[91] Thusoo, Ashish, Sarma, Joydeep Sen, Jain, Namit, Shao, Zheng, Chakka, Prasad, Anthony, Suresh, Liu, Hao, Wyckoff, Pete, and Murthy, Raghotham. Hive - a warehousing solution over a map-reduce framework. *PVLDB 2*, 2 (2009), 1626–1629.

[92] Toshniwal, Ankit, Taneja, Siddarth, Shukla, Amit, Ramasamy, Karthik, Patel, Jignesh M., Kulkarni, Sanjeev, Jackson, Jason, Gade, Krishna, Fu, Maosong, Donham, Jake, Bhagat, Nikunj, Mittal, Sailesh, and Ryaboy, Dmitriy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2014), SIGMOD '14, ACM, pp. 147–156.

[93] Vavilapalli, Vinod Kumar, Murthy, Arun C., Douglas, Chris, Agarwal, Sharad, Konar, Mahadev, Evans, Robert, Graves, Thomas, Lowe, Jason, Shah, Hitesh, Seth, Siddharth, Saha, Bikas, Curino, Carlo, O'Malley, Owen, Radia, Sanjay, Reed, Benjamin, and Baldeschwieler, Eric. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (New York, NY, USA, 2013), SOCC '13, ACM, pp. 5:1–5:16.

[94] White, Tom. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., June 2009.

[95] Yang, Fan, Qian, Zhengping, Chen, Xiuwei, Beschastnikh, Ivan, Zhuang, Li, Zhou, Lidong, and Shen, Jacky. Sonora: A platform for continuous mobile-cloud computing. *Technical Report, Microsoft Research Aisa* (2012).

[96] Yang, Hung-chih, Dasdan, Ali, Hsiao, Ruey-Lung, and Parker, D. Stott. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2007), ACM, pp. 1029–1040.

[97] Yu, Yuan, Gunda, Pradeep Kumar, and Isard, Michael. Distributed aggregation for data-parallel computing: interfaces and implementations. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (New York, NY, USA, 2009), ACM, pp. 247–260.

[98] Zaharia, Matei, Chowdhury, Mosharaf, Das, Tathagata, Dave, Ankur, Ma, Justin, McCauley, Murphy, Franklin, Michael J., Shenker, Scott, and Stoica, Ion. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 2–2.

[99] Zaharia, Matei, Das, Tathagata, Li, Haoyuan, Hunter, Timothy, Shenker, Scott, and Stoica, Ion. Discretized streams: fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 423–438.

[100] Zeitler, Erik, and Risch, Tore. Massive scale-out of expensive continuous queries. *PVLDB 4*, 11 (2011), 1181–1188.

[101] Zou, Qiong, Wang, Huayong, Soulé, Robert, Hirzel, Martin, Andrade, Henrique, Gedik, Buğra, and Wu, Kun-Lung. From a stream of relational queries to distributed stream processing. *Proc. VLDB Endow. 3*, 1-2 (Sept. 2010), 1394–1405.