Managing Skew in the Parallel Evaluation of User-Defined Operations

YongChul Kwon

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

2012

Reading Committee:

Magdalena Balazinska, Chair

Bill Howe

Dan Suciu

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Managing Skew in the Parallel Evaluation of User-Defined Operations

YongChul Kwon

Chair of the Supervisory Committee:
Professor Magdalena Balazinska
Computer Science and Engineering

Science and business are generating data at an unprecedented scale and rate due to ever evolving technologies in computing and sensors. Analyzing big data has become a key skill driving business and science. The challenges in big-data analysis stem not only from the data volume, but also from the diversity of data types to analyze (*e.g.*, text, image, audio, video, and graph) and the various analyses beyond relational algebra that need to be performed (*e.g.*, machine learning, natural language processing, image processing, and graph analysis). The user-defined operation (UDO) is a powerful mechanism to implement complex data processing tasks without changing the core of the parallel data processing engine. Although users can rapidly develop a new data analysis task with UDOs and execute the task in a cluster of computers, achieving high performance is important for users, especially those who do not have an extensive background in programming.

This thesis focuses on addressing skew in parallel UDO evaluation. Skew is a problem when there exists a significant variance in the execution time of parallel tasks. In the presence of skew, the benefit of using a parallel system diminishes. Our detailed case study demonstrates that a new data analysis task can be rapidly implemented in a MapReduce-like system, but such implementation may be prone to skew problem during execution. A skew-resilient implementation is possible but requires significant implementation effort and expertise in programming. We also analyze the skew problem in three real workloads and show that skew problem is frequent (more than 40% of long running jobs experience skew).

The thesis proposes two techniques to manage skew in parallel UDO evaluations: SkewReduce and SkewTune. SkewReduce is a static data partition optimization technique for feature-extracting applications that are common in scientific analysis. SkewReduce can improve the application runtime by up to 8x compared with a default MapReduce data partitioning strategy without any code-level optimization. SkewTune is a transparent dynamic skew mitigation technique for MapReduce applications. SkewTune can improve the application runtime by up to 4x compared with default MapReduce engine without modifying the application source code, without requiring any input from the developer or user, and without causing any side-effect during the execution.

TABLE OF CONTENTS

Page

List of Figures iii

List of Tables v

Chapter 1: Introduction 1
1.1 The Need for Parallel User-Defined Operation (UDO) Execution 2
1.2 Challenges of Parallel UDO Execution 3
1.3 Contributions of the Thesis 6
1.4 Outline of Thesis 10

Chapter 2: Motivating Example 11
2.1 Friends of Friends Clustering Algorithm 12
2.2 Basic Distributed Friends of Friends 14
2.3 Scalable Distributed Friends of Friends 19
2.4 Implementation 22
2.5 Evaluation 24
2.6 Conclusion 34

Chapter 3: Study of Skew in MapReduce Applications 35
3.1 MapReduce Programming Model 35
3.2 Types of Skew in a MapReduce Application 38
3.3 Skew in the Real World 46
3.4 Best Practices 61
3.5 Conclusion 63

Chapter 4: SkewReduce: Cost-based Partition Optimization 65
4.1 Feature Extracting Applications 67
4.2 SkewReduce 71
4.3 Evaluation 84
4.4 Conclusion 96

Chapter 5: SkewTune: Dynamic Skew Mitigation 97
5.1 SkewTune Design Requirements 99
5.2 SkewTune Approach 101
5.3 SkewTune for Hadoop 115
5.4 Evaluation 118
5.5 Conclusion 126

Chapter 6: Related Work 128
6.1 Parallel UDO Evaluation Systems 128
6.2 Skew Handling through Skew-Resilient Implementation 129
6.3 Skew Handling for UDOs 133

Chapter 7: Conclusion and Future Work 136
7.1 Short-Term Future Work 138
7.2 Long-Term Future Work 139
7.3 Final Remarks 141

Bibliography 142

LIST OF FIGURES

Figure Number Page

1.1 Example of Skew in a MapReduce Application 4

2.1 Friends of Friends clustering algorithm 13
2.2 Dataflow in dFoF algorithm 15
2.3 Illustration of the dFoF algorithm 17
2.4 Uniform partitioning and Non-uniform partitioning 22
2.5 Example Dryad execution plan for dFoF 23
2.6 Average time to cluster entire snapshot 27
2.7 Runtime breakdown across phases 28
2.8 Distribution of FoF runtime per partition 29
2.9 Distribution of FoF peak memory utilization per partition 30
2.10 Speedup of OpenMP dFoF and Dryad dFoF 31
2.11 Scaleup of OpenMP dFoF and Dryad dFoF 32

3.1 Overview of an execution of a MapReduce job in Hadoop 36
3.2 Distribution of task runtimes for PageRank 39
3.3 Distribution of task runtime for CloudBurst 41
3.4 Runtime distribution of the local clustering phase of the Friends-of-Friends algorithm 43
3.5 Distribution of the number of key groups and input records per reduce task for CloudBurst 44
3.6 Cumulative fraction of straggler tasks 50
3.7 Cumulative fraction of ratio of straggler runtime to median task runtime . . . 51
3.8 Distribution of map runtime with respect to # of input records per application 53
3.9 Distribution of reduce runtime with respect to # of reduce keys per partition function 55
3.10 Fraction of jobs per partition function 57
3.11 Fraction of speculation result per phase per cluster 59
3.12 Successful speculative execution 60

4.1 A scatter plot of flow cytometry measurements 68

4.2 Illustration of the merge step of the clustering algorithm in the SkewReduce framework . . . 75
4.3 Relative runtime of different partitioning strategies compared with the SkewReduce optimized plan . . . 88
4.4 Completion times of plans for the Astro dataset using different cost functions 90
4.5 Completion time of plans for Seaflow dataset using different cost functions . . 91
4.6 Completion time for the Astro dataset with varying sample rates . . . 92
4.7 Completion time for the Seaflow dataset with varying sample rates . . . 93
4.8 Optimization time with varying sample rates and cost functions . . . 94
4.9 Aggregate output data size produced at each level of an SkewReduce partition plan . . . 95

5.1 Conceptual skew mitigation in SkewTune . . . 102
5.2 Merging Result of Parallel Scan . . . 110
5.3 SkewTune Architecture . . . 115
5.4 UDO runtime with and without SkewTune. . . . 120
5.5 Performance Consistency of Map Phase . . . 123
5.6 Runtime of Map Phase without Skew . . . 124
5.7 Runtime of Reduce Phase without Skew . . . 125
5.8 Overhead of Local Scan vs. Parallel Scan . . . 127

LIST OF TABLES

Table Number Page

2.1 Distribution of the number of particles within distance threshold 25

3.1 Summary of Analyzed Workloads 47
3.2 Overall Statistics for Straggler Analysis 48
3.3 The number and fraction of jobs that have straggler tasks 49
3.4 Types of Map Functions in Figure 3.8 54
3.5 Types of Partition Functions in Figure 3.9 56
3.6 Total number of jobs and speculated tasks and default setting for speculative execution. 58

4.1 Notations in Section 4.2 71
4.2 Datasets used in the evaluation 86
4.3 Cost-to-time conversion constant for cost models (ρ_p, ρ_m,scale) 86
4.4 Cost functions for evaluation 89

5.1 Notations in Section 5.2 103
5.2 Mitigation Overhead Statistics 126

ACKNOWLEDGMENTS

I am closing the longest chapter of my life so far. My graduate career has been a time of meeting great people, discussing exciting ideas and learning how to conduct research. Of course, there were some hard times, but I also learned how to get the most from them through my career. Here, I want to share my sincere gratitude and appreciation for those who played a vital roles in my graduate career.

I am forever in debt to Magdalena Balazinska for her advising, support, and patience. She is a fantastic adviser who guides my research and a friend on whom I can rely. The two greatest things that I learned from her are optimism and patience, in which I was lacking before starting my graduate career.

I thank Bill Howe and Dan Suciu for their support, advice, and excellent feedback on my research. Their perspectives on research are always inspiring. I thank Dan Grossman and Virginia Armbrust for so enthusiastically being being on my committee.

I thank all my coauthors Jeff Gardner, Jerome Rolia, and Kai Ren. They are all great and hard-working collaborators and hard working, so hard that I am often ashamed of my procrastination. I hope that I have a chance to meet Jerome and Kai in person sometime.

I thank the University of Washington Database Group for inspiring discussions, feedback, and friendship: Mike Cafarella, Nilesh Dalvi, Wolfgang Gatterbauer Abhay Jha, Nodira Khoussainova, Paris Koutris, Julie Letchner, Alexandra Meliou, Kristi Morton, Vibhor Rastogi, Chris Ré, Marianne Shaw, Emad Sourush, Prasang Updhyaya, Jingjing Wang, Evan Welbourne, and Shenliang Xu.

I thank the University of Washington astronomy group for their support and collaboration. The motivation for this thesis came from early collaborations with them: Yusra AlSayyad, Andrew Connolly, Jeff Gardner, Simon Krughoff, Sarah Loebman, Tom Quinn, and Keith Wiley.

I thank Eytan Adar, Waylon Brunette, Krishnamurthy Dvijotham, Svetoslav Kolev, Yang Li, Thomas Lin, and Supasorn Suwajanakorn for being friends and good office mates. I will not forget those exciting chats on so many different topics.

I thank my friends: Daihyun Baik, Ivan Beschastnikh, Taeshik Earmme, Kyungnam Han, Seungyeop Han, Susumu Harada, Alan Ho, Miryung Kim Ho, Sukpyo Hong, Jaeyeon Jung, Felix Sunjoo Kim, Jintae Kim, Wanki Kim, Seongjae Lee, Suin Lee, Taehee Lee, Yongjoon Lee, Sangjoon Park, Kayur Patel, Hoifung Poon, Fei Wu, and Hyuntae Yoo. Without them, my life in Seattle would have been gloomy (even more in winter). I thank Jungwoo Ha and Changkyu Kim for their helpful advice during my job search.

I thank Dr. Sejune Hong and Prof. Junehwa Song for their encouragements to pursue graduate career. I also thank Lindsay Michimoto and Andrew Petersen. Andrew took me under his wing during my first year and helped me successfully settle successfully into the department. Lindsay took over Andrew's role after he left to pursue his career. She has always been a valuable source of information to which I often turned to survive in the graduate school.

Finally, I thank my family. I deeply thank my parents for their unconditional love and support throughout my life. I thank my wife, Yoon Jee Cho, for her support, patience, trust, and love. Marrying her is my greatest personal achievement in my graduate career!

Chapter 1

INTRODUCTION

Science and business are becoming data-intensive [2, 64, 99, 126]. Computerized services, advanced sensors, and ubiquitous devices let companies, research communities, and individual research groups collect data at unprecedented rates and resolutions. In astronomy, the recently constructed Large Synoptic Survey Telescope (LSST) will cover half the sky every three nights, detecting 5.7×10^8 sources and generating over 30 TB of data per night [94]. In physics, the Large Hadron Collider is generating approximately 25 PB of data every year [141]. In biology, recent high throughput sequencing devices can generate millions of DNA sequence reads, and the European Bioinformatics Institute (EBI) increased storage for those sequence reads from 2.5 PB in 2008 to 5 PB in 2009 [121]. In neuroscience, electron microscopy over large cortical volume is expected to produce 10 TB of volumetric data on fine-scale brain anatomy [88]. In 2010, Facebook stored 15 PB of data in its data warehouse and has loaded 60 TB of new data every day [132]. In May 2010, Google processed more than 946 PB of data using the MapReduce system [36, 37].

Managing and analyzing data at a large scale has become a key skill driving business and science. Big-data management and analysis are predicted to be a $50 B industry by 2017, a growth factor of 10 from the $5 B market size as of 2012 [71]. There are two prominent types of systems in big-data analysis: parallel database management systems (DBMSs) and MapReduce-type systems. A parallel DBMS is highly optimized to run relational queries in a cluster of computers [58, 101, 124, 128, 137]. For expensive relational operations, such as join and group by, parallel DBMSs implement several different algorithms that work best under different conditions. These engines select the most promising combination of strategies for each query submitted to the system. MapReduce [37] and its open source implementation Hadoop [61] are also popular in big-data analysis. MapReduce provides simple yet general programming and execution models for distributed data analy-

sis applications. In MapReduce, relational queries are supported by high-level abstraction layers that translate queries written in high-level languages into a directed-acyclic graph of MapReduce jobs [18, 27, 104, 109, 130, 147].

1.1 The Need for Parallel User-Defined Operation (UDO) Execution

In addition to relational operators, RDBMSs and their parallel counterpart parallel DBMSs have supported user-defined operations (UDOs) as mechanisms to extend the functionality of the engine [45, 16, 97, 106, 110]. Through UDOs, users can incorporate new data types and functions that are not included in the DBMS by default. In MapReduce-type systems, UDO is the default mode of operation; relational operators are implemented as UDOs.

The need for UDOs in big-data systems is accelerated today by two emerging trends.

- **Complex Extract-Transform-Load.** Unstructured data (*e.g.*, text, audio, image, and video) are a gold mine of information [28, 92]. Before data analysis, unstructured data often require a complex extract-transform-load (ETL) process. For example, in modern sentiment analysis, blog posts, tweets, and Facebook wall posts are first treated by a natural language processing (NLP) parser that parses each sentence and tags each word with a part of speech [78]. The tagged sentences are then analyzed to extract a sentiment. Running image, audio, and video analysis often requires complex preprocessing (*e.g.*, ETL) such as color normalization and various transformations (*e.g.*, sharpen, blur, and spectral). Such complex operations are not built in the RDBMSs. They thus have to be implemented as UDOs. For example, the *New York Times* uses Hadoop [61] with Amazon EC2 for large-scale image conversion [10]. Astronomers use Hadoop to preprocess telescope images for advanced analyses such as anomaly detection, classification, and moving-object tracking [139].
- **Advanced analysis beyond relational algebra.** Relational algebra and its close derivatives are not enough to support complex machine learning algorithms and complex analysis over non-relational data types (*e.g.*, graph, temporal, spatial, and spatio-temporal). They are also insufficient to scale legacy data-analysis implementations [9, 26, 74, 85, 95, 127]. However, ad hoc development of a new data analysis software to

efficiently process petascale data is difficult and expensive [15]. Both parallel DBMSs and MapReduce-type systems provide the following benefits: (1) they run on inexpensive shared-nothing clusters; (2) they provide quick-to-program, declarative interfaces with support for UDOs [18, 27, 69, 104, 109, 130, 147]; and (3) they manage all task parallelization, execution, and failure-handling challenges. Thus, these frameworks hold the promise of enabling cost-effective, massive-scale data analysis with low development costs.

Thus, complex operations over today's big-data increasingly require the ability to execute complex UDOs in parallel.

1.2 Challenges of Parallel UDO Execution

DeWitt and Gray identified three factors that threaten the scalability of a parallel DBMS: *startup*, *interference*, and *skew* [42]. For convenience, we use the term *job* to refer either to a SQL query or a MapReduce job. A job is represented as a directed acyclic graph (DAG) of operators (*e.g.*, a relational algebra plan or a pair of map and reduce operations). Each operator is executed by distributed *tasks* in a cluster of compute nodes. Each task corresponds to a partition of the UDO.

We briefly review each factor then concentrate on *skew*, the main focus of this dissertation.

- **Startup** refers to the overhead of distribution and initialization of a new job. For a short job, the time to transfer the job specification to each compute node may dominate the actual processing time. The fraction of overhead in the job execution decreases as the job runs longer.
- **Interference** occurs when concurrent jobs (or tasks) in the system interfere with each other due to contention in accessing shared resources such as network, disk, and memory bandwidth. When interference occurs, adding a new task to the system does not improve throughput or latency after a certain number of tasks are already running.

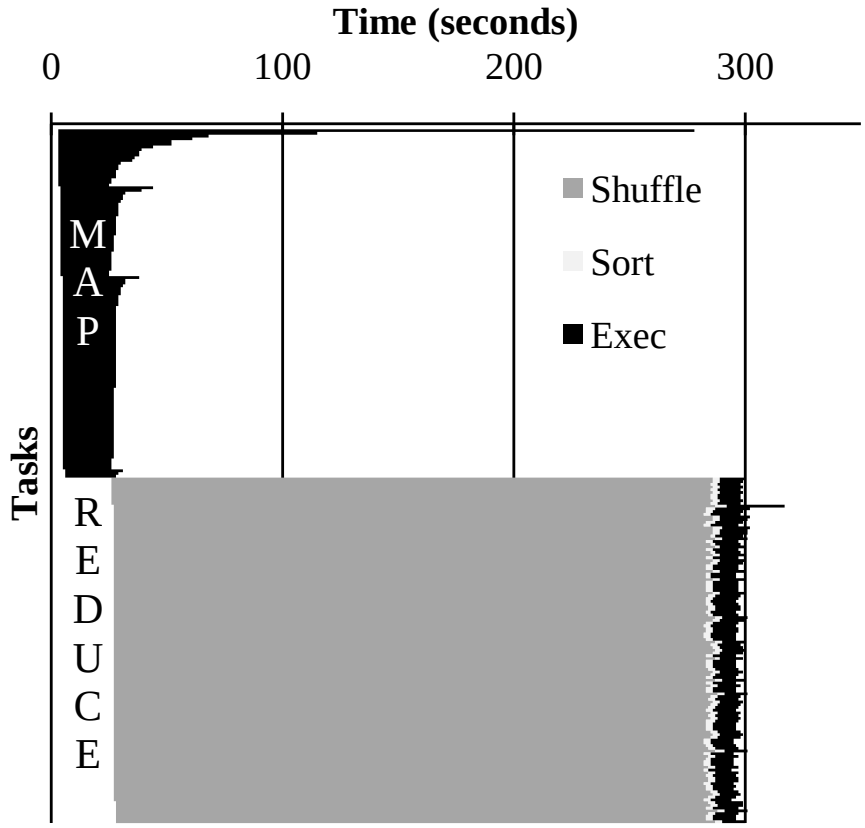


Figure 1.1: A timing chart of a MapReduce job running the PageRank algorithm from Cloud 9 [72]. Exec represents the actual map and reduce operations. The slowest map task (first one from the top) takes more than twice as long to complete as the second slowest map task, which is still 5x slower than the average. If all tasks took approximately the same amount of time, the job would be completed in less than half the time.

- **Skew** refers to a significant variance in task runtime. With greater skew, the benefit of parallelization diminishes because the completion time of a query is dominated by the slowest task. Figure 1.1 illustrates the skew problem in a MapReduce job. We use PageRank [25] as an example of a UDO. As the figure shows, this UDO is expressed as a MapReduce job, which runs in two main phases: the map phase and the reduce phase. Load imbalance can occur either during the map or reduce phases. We refer to such an imbalanced situation as *map-skew* and *reduce-skew*, respectively.

Skew can lead to significantly longer job-execution times and significantly lower cluster throughput. In the figure, each line represents one task. Time increases from left to right. This job exhibits map-skew: a few map tasks take 5x to 10x as long to complete as the average, causing the job to take twice as long as an execution without skew.

The startup and interference factors are relatively independent of the semantic of applications or operations. They can thus be addressed by system level optimizations such as caching of UDO binaries or a better task scheduling algorithm. In contrast, skew is closely related to the semantics and the implementation of the operator (*i.e.*, how the data is processed) because both the data and the implementation affect the runtime of an operator (we present more details with examples in Chapter 2 and 3). It is possible to develop a skew-resilient implementation (*i.e.*, it produces the correct result and does not experience skew at runtime for all input data), but such implementation may vary between operators since each operator has distinct semantics. There are two problems that make it harder to handle skew with a UDO than with relational operators.

First, a UDO is a black box to the execution engine. For the relational operators such as join and aggregate, the semantics are well understood, and many specialized techniques to handle skew problems are available for those operators [42, 66, 77, 87, 117, 118, 142, 143, 144, 145]. In contrast, virtually nothing of a UDO is known to the execution engine except the type signature (and maybe some high-level hints if the user is willing to provide them). Also, it is not feasible to request users to devise techniques and optimize their code to overcome skew problems. For example, as shown in Chapter 2, troubleshooting an application consisting of UDOs can take several weeks and require a deep understanding not only of the application domain but also of the computer systems, algorithms, data structures, and programming.

Second, a suboptimal implementation is likely to be the norm for UDOs, especially for MapReduce-type systems that are designed to run UDOs. MapReduce systems have a rapidly growing user base in various domains including small IT startups, large IT companies such as eBay [44], Facebook [47], Twitter [133], and domain scientists from universities [10, 114]. According to Ren *et al.*, many users of Hadoop are writing their applications

in traditional programming languages [114]. Given such a diversity of users, it is clear that not all users are professional developers. We cannot safely assume that a UDO is as highly optimized as relational operators. The source of the problem is ranges from poor algorithm design (*e.g.*, a user can easily implement a $O(n^2)$ sort algorithm where $O(n \log n)$ is possible) to suboptimal engineering decisions (*e.g.*, poor choice of data structure) to poor understanding of the data (*e.g.*, uniformity assumption on data distribution). In a recent study of three research Hadoop clusters, we found that more than 50% of submitted jobs involved user-defined functions in all clusters [114]. Also, more than 40% of jobs that were running longer than five minutes had at least one task that experienced skew (more details in Section 3.3). Thus, unlike relational operators for which skew is well understood and effectively mitigated [42, 87, 117, 118, 142, 143, 144, 145], UDOs today commonly suffer from skew.

1.3 Contributions of the Thesis

MapReduce [37] has proven itself as a powerful and cost-effective approach for writing UDOs and applying them to massive-scale datasets [10]. MapReduce provides a simple API for writing UDOs; a user only needs to specify a serial map function and a serial reduce function. The implementation takes care of applying these functions in parallel to a large dataset in a shared-nothing cluster. MapReduce has influenced UDO API of many other systems [35, 48, 58, 128, 137] and become a popular low-level execution layer for high-level query languages [18, 27, 104, 109, 130, 147]. For this reason, in this dissertation, we focus on UDOs implemented in the context of a MapReduce-type system.

This thesis considers the problem of scaling parallel UDO evaluations in the presence of skew. We focus on MapReduce systems because they are popular and general and the problem is potentially more severe than in a parallel DBMS because of diverse users and UDO intensive workloads [114]. However, the ideas and techniques proposed in the thesis can be generalized to other types of parallel execution engines, including parallel DBMSs.

The high level contribution of the thesis is threefold. First, the thesis presents a case-study and a measurement study of MapReduce applications to understand the skew problems in parallel UDO evaluations (Chapters 2 and 3). Second, the thesis proposes a static

data partition optimization technique called SkewReduce for feature-extracting applications that are popular in scientific data analysis and image processing (Chapter 4). The last contribution is a dynamic skew mitigation technique called SkewTune for general MapReduce applications (Chapter 5).

The detailed main contributions of the thesis are:

- **A Case Study of Scaling Data Analysis using MapReduce**: We ask, how hard is it to convert and optimize a legacy data analysis application to a MapReduce application? We present a detailed case study of scaling an existing data analysis algorithm using a MapReduce system. We obtained the pseudocode of a clustering algorithm from astronomers [33] and implemented the algorithm in MapReduce through a line-by-line translation using off-the-shelf data structures. We also obtained two test datasets from astronomers for the evaluation. Although the two datasets have exactly the same numbers of bytes and items, surprisingly there is a factor of 20 difference in job completion time due to skew. In Chapter 2, we present the investigation and application-specific optimization efforts and show that addressing skew problem requires significant effort and experience in algorithm design, data structures, programming, and even distributed systems.
- **A Survey of Skew in MapReduce Applications:** We ask, is the skew problem general or unique to the astronomy application? Are there a few common causes of the skew problem in MapReduce applications? How frequently does the problem arise in real clusters and how significant is the problem? We generalized the experience with the one astronomy use-case through a measurement study of skew in existing MapReduce applications. To understand the skew problem better, we first analyzed execution logs of three different Hadoop MapReduce clusters and examined the skew problems that arose in those clusters. We also analyzed the details of several real world MapReduce applications with real world datasets then analyzed the skew problems observed during the execution of these applications. We categorize the observed skew problems into five different causes and create a taxonomy. We present five best practices in writing MapReduce applications to minimize the skew problem.

- **Static Skew Mitigation for Feature-Extracting Applications:** We ask, if the UDOs are susceptible to data skew and the best way to partition the data to avoid skew depends on the input dataset and the cluster configuration, can we automatically find a good data partitioning plan for a data set before execution? SkewReduce precisely answers this question for the feature-extracting applications that are popular in many domain sciences and image processing. The key idea behind SkewReduce is to ask the user for extra information about their UDOs. The user first creates cost functions that characterize the UDO runtime. Given the cost models and a sample of the input data, SkewReduce searches a good partition plan of the input data in the presence of data skew instead of relying on a fully optimized UDO implementation. For the domain experts such as scientists and statisticians, we posit that writing a cost model is easier than debugging and optimizing programs. Also, they can leverage their domain knowledge in the cost model. The cost models are specific to the implementation and can be reused across many data sets for the same UDOs.

  SkewReduce optimizes how data is partitioned across nodes with given a sample of the input data, the user-defined cost models, and the cluster configuration (*e.g.*, the number of nodes and the scheduling algorithm). The user-defined cost models estimate the cost of processing the entire dataset based on the data samples. Using the cost models, the SkewReduce optimizer searches a good degree of parallelism by a) applying finer grained data partitioning if a significant data skew is expected for a part of the input data, b) keeping coarse grained partitions when the data skew is not anticipated, and c) balancing the partition granularity in a way that minimizes expected job completion time under the given cluster configuration.

  To show the effectiveness of SkewReduce, we built a prototype running on top of Hadoop. From the evaluation, the partition optimization takes several minutes on a desktop machine. Compared to the default Hadoop setup, the runtime improves by factors of two to eight, depending on the dataset, without any code-level optimization. Another intriguing result is that, even with a naïve cost model (the number of records per partition), SkewReduce still yields better runtime than the default setup due to optimization that partitions the data as evenly as possible.

- **Dynamic Skew Mitigation in MapReduce:** We ask, can we take the SkewReduce idea further and build a system that mitigates skew without any input from the user whatsoever? Can the system accelerate the job experiencing data skew by exploiting idle nodes or dynamically adding nodes to the cluster?

  SkewTune is a dynamic technique to mitigate the data skew or to accelerate job execution by fully leveraging idle nodes. To automate the mitigation, SkewTune assumes that each UDO invocation is independent. This property must be ensured by the user. Also, SkewTune is designed for systems where the UDOs read the input from disks and write the output to disks.

  SkewTune continuously monitors the execution of UDOs and detects the current bottleneck task that dominates and delays the job completion time. Once such a task is identified, the task is stopped, and its unprocessed input data are repartitioned among idle nodes. The repartition only occurs when there is an idle resource (*i.e.*, The MapReduce scheduler runs out of other tasks to schedule) or when the nodes are dynamically added to accelerate the job (*e.g.*, using spot instances in Amazon EC2 [6]). Thus, if there are no tasks experiencing a significant data skew, SkewTune imposes no overhead. If there is an idle node and there is the current bottleneck task, SkewTune will stop the task, repartition the remaining input data with respect to future node availability, and parallelize processing. SkewTune continuously detects and removes the bottlenecks until the job completes.

  When repartitioning a task, SkewTune takes the future node availability of the cluster into account to minimize the job completion time. The availability is estimated from the progress of running tasks. The remaining unprocessed input data of a task is repartitioned in a way that fully utilizes nodes that are available immediately or in the near future. SkewTune also minimizes the side-effect of repartitioning so that the original output can be reconstructed by concatenating the output of split tasks.

  For the evaluation, we implemented the SkewTune strategy in the Hadoop MapReduce engine. Experimental results show that SkewTune can significantly improve completion time by up to factor of four without code change if there is a significant data skew. Also, it imposes, at most, a one-minute overhead due to estimation error in the

remaining time when the load is well balanced.

All surveys and systems are evaluated using real-world datasets and real-world applications most of which are available online [129]. The prototypes of SkewReduce and SkewTune are available as open-source software [82, 83].

1.4 Outline of Thesis

The thesis is organized as follows: we first show the motivation of the thesis by presenting a case study of scaling a clustering algorithm using Dryad, a parallel dataflow engine [68] (Chapter 2). We then survey the skew problems in MapReduce applications (Chapter 3).

We present two approaches that mitigate skew in parallel UDO evaluation (Chapter 4 and 5). The SkewReduce system addresses skew problem in a feature-extraction application through static data partitioning prior to execution (Chapter 4). The SkewTune system addresses the skew problems through late skew detection and proactive data repartitioning at runtime for general MapReduce applications (Chapter 5). We present related work (Chapter 6), and then conclude with future research directions (Chapter 7).

Chapter 2

MOTIVATING EXAMPLE

In this chapter, we present a case study where we implement an existing data analysis application as a series of user-defined operations and execute it at scale using a modern parallel data processing system. The application is a clustering algorithm called friends-of-friends (FoF) used in computational astrophysics [33]. The parallel data processing system we used is Dryad [68], a distributed dataflow engine that provides a more general interface than MapReduce. The application is written in DryadLINQ, a high-level layer on top of Dryad [147].

The astronomers use the FoF algorithm to quantify the structure in a snapshot of an N-body simulation that simulates the evolution of the universe from shortly after the Big Bang to the present day [100]. Due to its simplicity, FoF is one of only two algorithms that have been implemented in a distributed parallel fashion in the astrophysics community [50, 51]. At the time of our study, there were two C implementations of the FoF algorithm. One version is implemented using Message Passing Interface (MPI) [120] and highly optimized using a custom distributed data structure [50, 51]. Another version is a multi-threaded OpenMP implementation that runs in a single address space [32]. That is, the OpenMP version can only handle snapshots that entirely fit in the memory of one machine. The optimized MPI version has the best performance at the cost of significant implementation efforts. The OpenMP version is easier to implement than the MPI version, but the input data size is limited by the memory size of a single machine.

In this chapter, we present dFoF, which is the implementation of FoF in DryadLINQ. The goal of the translation was to demonstrate the feasibility of expressing a sophisticated data analytics task from the astronomy domain using the API of an off-the-shelf big-data processing system. Analyzing the output of N-body simulations is difficult today and the community could benefit from the ability to carry out such analyses without manually pro-

gramming each question in MPI. Thanks to the simple programming model of DryadLINQ, we could develop dFoF from the pseudocode in several weeks by only implementing the core of the algorithm. Thanks to the great scalability of the Dryad engine, we could process large datasets in a cluster of commodity hardware. Unlike the OpenMP version, dFoF does not require a shared memory machine that has enough memory to hold the entire dataset. dFoF also achieves great scaleup and speedup as we add more nodes to the cluster.

The case study motivates this thesis work on skew mitigation in parallel user-defined operations. We show that a naïve implementation of dFoF that we developed only in a couple of days is prone to skew. One dataset took 20x longer to process than another dataset even though the two datasets had exactly the same number of tuples. We also show that a skew-resistant implementation (*i.e.*, the implementation takes a similar amount of time to process any dataset with the same number of tuples) is possible but developing such an implementation took several weeks of testing, diagnosis, and optimizations that all required expertise in algorithm, data structure, and programming.

2.1 Friends of Friends Clustering Algorithm

Application domain. Cosmological simulations serve to study how structure evolves in the universe on distance scales ranging from a few million light-years to several billion light-years in size. In these simulations, the universe is modeled as a set of particles. These particles represent gas, dark matter, and stars and interact with each other through gravitational force and fluid dynamics. Particles may be created or destroyed during the simulation (*e.g.*, a gas particle may spawn several star particles). Every few simulation timesteps, the program outputs a snapshot of the universe as a list of particles, each tagged with its identifier, location, velocity, and other properties. The data output by a simulation can therefore be stored in a relation with the following schema:

$$Particles(id, x, y, z, v_x, v_y, v_z, mass, density, \cdots)$$

State of the art simulations (*e.g.*, Springel *et al.* [122]) use over 10 billion particles producing a dataset size of over 200 GB per snapshot. The NCSA/Cray Blue Waters system [21] supports astrophysical simulations that generate 100 TB per snapshot and a

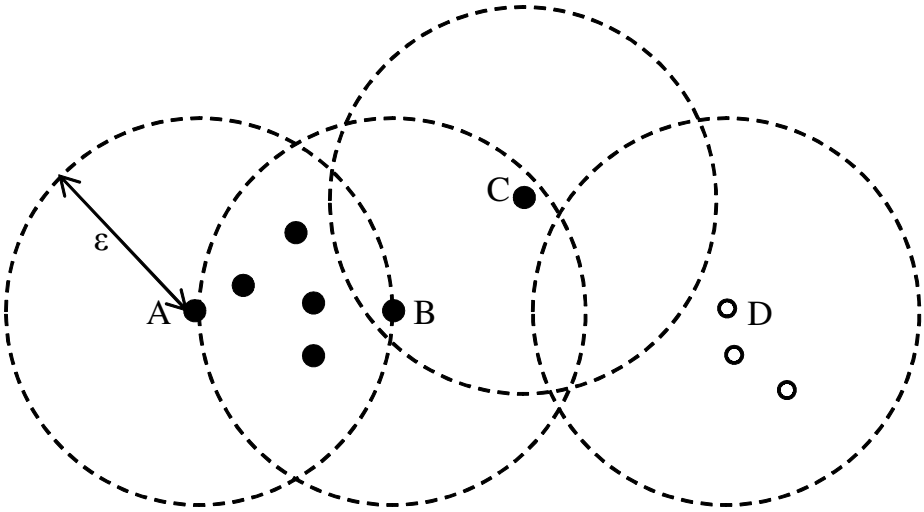


Figure 2.1: **Friends of Friends clustering algorithm.** Two particles are considered *friends* if the distance between them is less than a threshold ϵ: A and B are friends and B and C are friends, but A and C are not. The friend relation is symmetric if the distance is symmetric. The friend of friend (FoF) relation is defined between two points if they are contained in the transitive closure of the friend relation (*e.g.*, A and C are a friend of friend pair via B). In the figure, the FoF relation induces a partition on the particles: all black points are in one cluster and all white points are in another.

total data volume of more than 10 PB per run.

Friends of Friends Clustering Algorithm. The friends-of-friends (FoF) algorithm (*c.f.*, Davis *et al.* [33] and references therein) has been used in cosmology for at least 20 years to identify interesting objects and quantify structure in simulations [113, 122]. FoF is a simple clustering algorithm that accepts a list of particles (pid, x, y, z) as input and returns a list of cluster assignment tuples $(pid, clusterid)$. To compute the clusters, the algorithm examines only the distance between particles. FoF defines two particles as friends if the distance between them is less than ϵ. Two particles are *friends-of-friends* if they are reachable by traversing the graph induced by the friend relationship. To compute the clusters, the algorithm computes the transitive closure of the friend relationship for each

unvisited particle. All particles in the closure are marked as visited and linked as a single cluster. Figure 2.1 illustrates this clustering algorithm. To find the friends of a particle, the algorithm initially builds a spatial index of the input particles then retrieves the friend particles with a range query on the spatial index for each particle in the input dataset.

2.2 Basic Distributed Friends of Friends

In this section, we introduce dFoF, our distributed FoF algorithm for MapReduce-style shared-nothing clusters. We discuss critical optimizations that make this algorithm truly scalable in the following section.

The basic idea behind any distributed clustering algorithm is to (1) partition the space of data to be clustered, (2) independently cluster the data inside each partition, and finally (3) merge clusters that span partition boundaries. There are several challenges related to implementing this type of algorithm on a MapReduce-style platform and in the context of astronomy data.

Challenges. First, in astrophysical applications, there is no characteristic cluster size or mass. The clustering of matter in the universe is largely scale-invariant at the size represented by the simulation. This means a cluster can be arbitrarily large and span arbitrarily many partitions. To identify such arbitrarily-large clusters from locally found ones, one cannot simply send to each compute node its own data plus a copy of the data at the boundary of adjacent partitions. Indeed, nearly all data would have to be copied to merge the largest clusters. Alternatively, one could try to use a global index structure, but this approach requires extensive inter-node communication and is therefore unsuitable for the dataflow-style processing of MapReduce-type platforms. In this chapter, we investigate a radically different approach. Instead of trying to use a distributed index, we redesign the algorithm to better follow the shared-nothing, parallel query processing approach and not require a global index at all. In this section, we present this algorithm, which we call dFoF.

Second, the uncharacteristic clusters pose a challenge for load balancing — each node needs to hold a contiguous region of space but there is no *a priori* spatial decomposition that is likely to distribute the processing load evenly. Load imbalances can negate the benefits of parallelism [42]. To achieve load balance and improve performance, we must

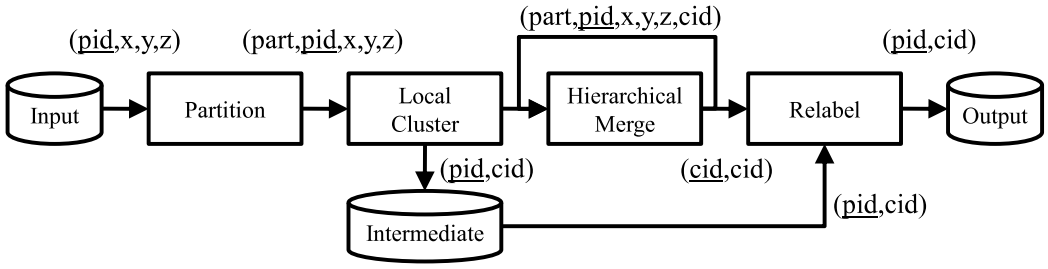


Figure 2.2: **Dataflow in dFoF algorithm.** dFoF runs in four phases. Each phase exchanges data in the form of a standard relation or set of key-value pairs. Underlined attributes are the primary keys of the corresponding relations. part represents a partition id. pid represents a particle ID. x, y, and z correspond to the particle coordinates. cid is a cluster id. Phases execute in series but with intra-phase parallelism.

ensure that each partition of the same operation processes its input data in approximately the same amount of time. This requirement is more stringent than ensuring each node processes the same amount of data. Indeed, in the FoF algorithm, execution times depend not only on the number of particles in a partition but also their distribution: small dense regions are significantly slower to process than large sparse ones. We discuss extensions to our algorithm that address these challenges in Section 2.3.

Approach. Our basic dFoF algorithm follows the typical distributed clustering approach in that the data is first partitioned, then clustered locally, and finally the local clusters are reconciled into large ones. Our algorithm differs from earlier work primarily in the way it handles the last phase of the computation. Instead of relying on a distributed index, dFoF reconciles large clusters through a *hierarchical merge process* that resembles a parallel aggregate evaluation [146]. To keep the cost of merging low, only the *minimum amount of data* is propagated from one merge step to the next. The rest of the data is written to disk before the merge. A final *relabeling* phase takes care of updating this data given the final merged output. dFoF thus runs in four phases: *Partition*, *Local cluster*, *Hierarchical merge*, and *Relabel*. Figure 2.2 shows the overall data flow of the algorithm, with each step labeled with the type of its output. We now describe the four phases in more

detail using a simple 2D example.

Partition. During partitioning, we assign each node a contiguous region of space to improve the probability that particles in the same cluster will be co-located on the same node. Figure 2.3 illustrates a 2D space split into four partitions P_1 through P_4. To determine these uniform regions, dFoF recursively bisects the space, along all dimensions, until the estimated number of particles per region is below the memory threshold of a node, such that local processing can be performed entirely in memory. We call the hierarchical regions *cells*, and the finest-resolution cells — the leaves of the tree — *unit cells*. The output of this phase is a partition of all data points (*i.e.*, particles).

Local Cluster. Once the data is partitioned, the original FoF algorithm runs within each unit cell. As shown in Figure 2.2, the output of this phase is written to disk and consists of a set of pairs: `(pid,cid)`, where `pid` is a particle ID and `cid` is a globally-unique cluster ID. Each input particle is labeled with exactly one cluster ID. Particles within distance ϵ of the boundary of each cell continue on to the next phase. They will serve to identify locally found clusters that need to be merged into larger ones.

Hierarchical Merge. To identify clusters that span multiple cells, particles at cell boundaries must be examined. If particles in adjacent partitions are within distance threshold ϵ of each other, their clusters must be merged. Figure 2.3 illustrates the merge step for four partitions P1 through P4. The outer boxes, P_i, represent the cell boundaries. The inner boxes, I, are distance ϵ away from the corresponding edge of the cell. In Figure 2.3(a), the local clustering step identified a total of six clusters labeled C1 through C6. Each cluster comprises points illustrated with a different shape and shade of gray. However, there are only three global clusters in this dataset. These clusters are identified during the hierarchical merge process. Clusters C3, C4, C5, and C6 are merged because the points near the cell boundaries are within distance ϵ of each other. Only points inside each P_i but outside each region I are needed to determine that these clusters must be merged. Figure 2.3(b) shows the actual input to the hierarchical merge following local clustering phase. This data reduction is necessary to enable nodes to process hierarchically larger regions of space efficiently

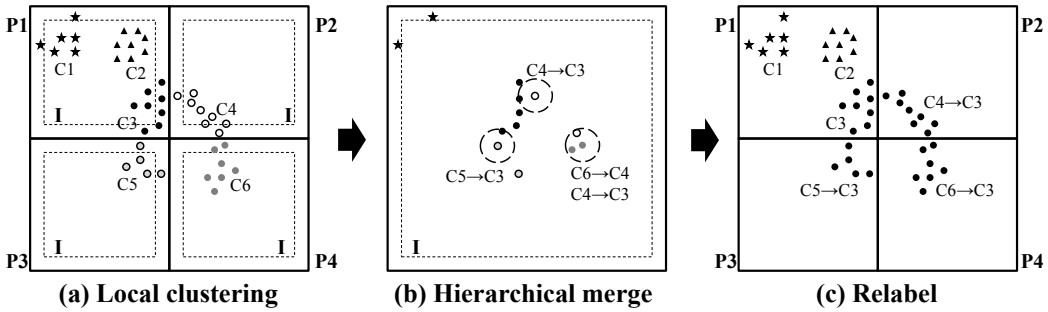


Figure 2.3: **Illustration of the dFoF algorithm.** (a) shows the first local clustering phase. Data is partitioned into four cells. Points with the same shape are in the same global cluster. Particles with different shades but with the same shape are in different local clusters. Each P_i shows the cell boundary and each I shows the interior region that is excluded during the *Hierarchical Merge* phase. (b) demonstrates *Hierarchical Merge* phase. Note that only boundary particles in (a) are considered during the merge phase. After the merge, three cluster mappings are generated: (C4,C3), (C5,C3), and (C6,C3). Such mappings are used to relabel local clusters during the *Relabel* phase as illustrated in (c).

and without running out of memory.

Algorithm 1, which we call `mergefof`, shows the detailed pseudocode of the merge procedure. At a high-level, the algorithm does two things. First, it re-computes the clusters in the newly merged space. Second, it relabels the cluster ids of those clusters that have been merged. The input is a set of particles, each labeled with a cluster id. The output is a set of pairs $(oldcid, newcid)$ providing a mapping between the pre-merge cluster ids and the post-merge cluster ids.

Lines 1-8 show the initial cluster re-computation whose output, M, is a nested set of clusters that must be merged. For example, for the dataset in Figure 2.3, M will have three elements, $\{\{C1\}, \{C3, C4, C5\}, \{C4, C6\}\}$. This set M, however, is not yet quite correct, as there are potentially members of M that should be further combined. To see why, recall that some particles — those in the interior of the merged regions — were set aside to disk before the merge process began. These set-aside particles may connect two otherwise disconnected

Algorithm 1 Merge result of FoF (`mergefof`)

Input: $D \leftarrow \{(pid, cid, x, y, z)\}$ // output from *Local Cluster* or *Hierarchical merge*
$\epsilon \leftarrow$ distance threshold
Output: $\{(old\ cid, new\ cid)\}$

1: $M \leftarrow \emptyset$ // nested set to store cluster ids of merged clusters
2: $R \leftarrow \emptyset$ // output mappings
3: $sidx \leftarrow$ `build_spatial_index`(D)
4: **for all** unvisited $p \in D$ **do** // compute cluster ids to merge
5: $N \leftarrow$ `friendclosure`$(p, \epsilon, sidx)$ // find all friends of friends of p using the spatial index
6: mark all $q \in N$ as visited
7: $C \leftarrow \{x.cid \mid x \in N\}$ // set of all cluster ids found in N
8: $M \leftarrow M \cup \{C\}$ // All cluster ids in N must be merged
9: **end for**
10: **repeat** // find additional clusters to merge
11: **for all** $C \in M$ **do**
12: $C^+ \leftarrow \{X | X \in M, C \cap X \neq \emptyset\}$
13: **if** $|C^+| > 1$ **then**
14: $M \leftarrow M - C^+$
15: $C' \leftarrow \{x | x \in X, X \in C^+\}$
16: $M \leftarrow M \cup \{C'\}$
17: **end if**
18: **end for**
19: **until** M does not change
20: **for all** $C \in M$ **do** // produce output
21: $newCid \leftarrow \min C$ // select the lowest identifier in C
22: $R \leftarrow R \cup \{(cid, newCid) | cid \in C\}$
23: **end for**
24: **return** R

clusters. In our example, C6 should be merged with C3, C4, and C5 but is not because the particles of C4 bridging C6 to C3 were set aside to disk before merging. We can infer such missing links by examining the pairwise intersections between sets of merged cluster identifiers. For example, since $\{C3, C4, C5\}$ and $\{C4, C6\}$ both contain C4, we infer that C3, C4, C5, and C6 are all part of the same cluster and can be assigned a single cluster id. The second step of `mergefof` (lines 10-19) performs this inference. In the last step, the

algorithm simply chooses the lowest cluster id as the new id of the merged cluster (lines 20-23).

Algorithm `mergefof` executes every time a set of child cells under the same parent are merged as we proceed up the cell hierarchy. After each execution, the mappings between clusters that are found are saved to disk. They will be reused during the final *Relabel* phase.

Relabel. In dFoF, there are two occasions for relabeling, *intermediate* and *global*. Intermediate relabeling assigns each particle used during the merge process a new cluster id based on the output of `mergefof`. This operation occurs once for each cell in the merge hierarchy. Global relabeling occurs at the end of dFoF. This operation first determines the final cluster ids for each local cluster id based on the accumulated output of `mergefof`. It then updates the local cluster assignments from the first phase with the final cluster id information by reprocessing the data previously set aside to disk as shown in Figure 2.2.

2.3 Scalable Distributed Friends of Friends

The dFoF algorithm presented thus far is parallel but not scalable due to skew effects. Some compute nodes during *Local clustering* phase may run significantly longer than others, negating the benefits of parallelism. In this section, we discuss this problem and present two optimizations that address it. The first optimization significantly improves the performance of both local `fof` and `mergefof` algorithms. The second optimization improves load balance.

2.3.1 Pruning Visited Subtrees

With an ordinary spatial index implementation, each partition can spend a significantly different amount of time processing its input during the local clustering phase (*i.e.*, FoF), despite having approximately the same amount of input data. We demonstrate this effect in Section 2.5, where we measure the variance in task execution times (in Figure 2.8, all plots except for non-uniform/optimized exhibit high variance). This imbalance is due to densely populated regions taking disproportionately longer to process than sparsely populated regions, even when both contain the same number of points.

To understand the challenge related to dense regions, the serial FoF algorithm computes the transitive closures of particles using repeated range-lookups in a spatial index as dis-

cussed in Section 2.1. The number of returned particles per lookup (*i.e.*, the traversed part of the index) are proportional to the density of the region. These lookups dominate the runtime. Astronomy simulation data is especially challenging in this respect, because the density can vary by several orders of magnitude from region to region[1]. To address this challenge, we optimize the local cluster computation as follows.

The original FoF algorithm constructs a spatial index over all points to speed up friend lookups. We modify this data structure to keep track of the parts of the subtree where all data items have already been visited. For each node in the tree (leaf node and interior node), we add a flag that is set to *true* when all points within the subtree rooted at the node have been returned as a result of previous friends lookups. The algorithm can safely skip such flagged subtrees because all data items within them have already been covered by previous lookups. By the nature of spatial indexes, points in a dense region are clustered under the same subtree and are therefore quickly pruned. With this approach, the index shrinks over time as the previously visited subtrees are pruned.

Because this optimization requires only one flag per node in the spatial index, it imposes a minimal space overhead. Furthermore, the flag can be updated while processing range lookups. In Algorithm 2, we show the modified version of the range search using this modified index structure. Line 5 is dependent on the type of spatial index. For a kd-tree, which we use in our prototype implementation, the condition can be evaluated by checking the flags of the child nodes and the data item assigned to the *root* node. For an R-tree, the condition can be evaluated similarly by checking child nodes and data items in a leaf node.

We apply this optimization both during the local clustering and the merging phases.

2.3.2 Non-Uniform Data Partitioning

While the above optimization solves the problem of efficiently processing dense regions, it does not solve all load imbalance problems. Indeed, with the uniform space-based partitioning described in Section 2.2, some nodes may be assigned significantly more data than others

[1]In the DBSCAN algorithm [46], the average complexity of a range lookup in a spatial index was assumed to be $\log n$ where n is the number of data items in the index. Clearly, this assumption does not always hold in FoF and its input data.

Algorithm 2 Range search with pruning visited subtree

Input: *root* ← search root node
query ← center of the range search (*i.e.*, querying object)
ϵ ← distance threshold
Output: set of objects within distance ϵ of *query*

```
1: if root.visitedAll is true then
2:     return ∅ // skip this subtree
3: end if
4: R ← ··· // normal range search for root
   // update bookkeeping information
5: if entire branch under root marked visited then
6:     root.visitedAll ← true
7: end if
8: return R
```

and may delay the overall execution or even halt if they run out of memory when the data is not uniformly distributed. The only way to recover is for the system to restart the job using a smaller unit cell. On the other hand, unit cells that are unnecessarily fine-grained add significant scheduling and merging overheads.

To address this challenge, we use a variant of Recursive Coordinate Bisection (RCB) scheme [17] to ensure that all partitions contain approximately the same amount of data (*i.e.*, same number of particles). The original RCB repeatedly bisects a space by choosing the median value along alternating axes to evenly distribute input data across multi-processors. Since the input data does not fit in memory, we first scan the data, collect a random sample at 1% sampling rate[2], and run RCB over the sample until the estimated size of the data for each bucket fits into the memory of one node. We use RCB because its spatial partitioning nature is well-suited to the underlying shared-nothing architecture (*i.e.*, it generates non-overlapping regions that are also easy to merge compared to, for example, space-filling curves). In Figure 2.4, we compare the uniform and data partitioning schemes. Because we use a sample instead of the entire dataset, there is some small discrepancy in the size of the

[2]If 1% sample does not fit in memory, we adjust the rate so that the entire sample fits in memory.

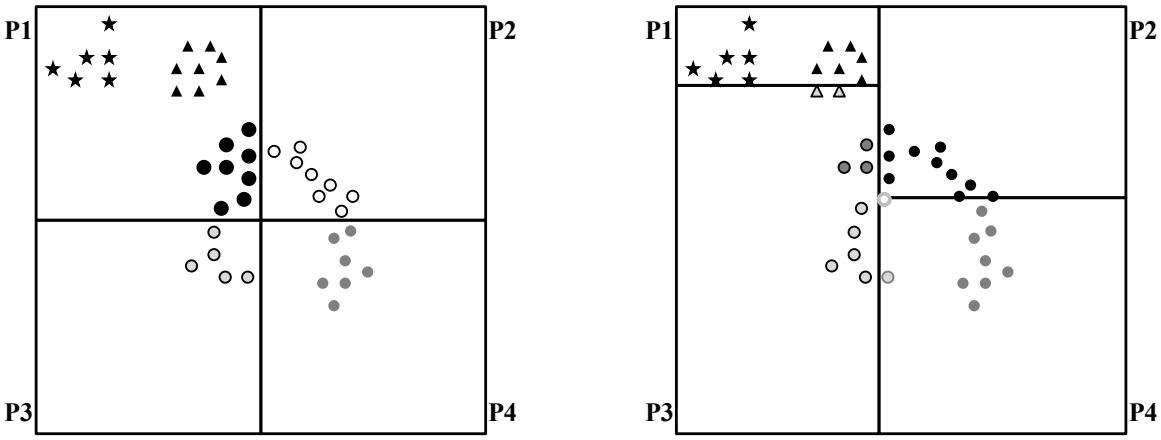


Figure 2.4: **Uniform partitioning and Non-uniform partitioning.** Uniform partitioning would generate uneven workloads: P_1 contains 22 points while P_3 have only 5 points in it. Data-oriented partitioning, however, produces an even workload: each partition is assigned 10 or 11 points.

partitions. Also, sampling requires an extra scan over the data, thus adding overhead to the entire job. However, it effectively reduces load skew, especially with the first optimization, and improves the job completion time as we show in Section 2.5.

2.4 Implementation

We implemented dFoF in approximately 3000 lines of C# code using DryadLINQ [147] the programming interface to Dryad [68]. Dryad is a massive-scale data processing system that is similar to MapReduce but offers more flexibility because its vertices are not limited to map or reduce operations. DryadLINQ is a Language-Integrated Query (LINQ) interface provider for Dryad. The LINQ offers relational-style operators such as filters, joins, and groupings and users can write complex data query succinctly and seamlessly in C#. At runtime, DryadLINQ automatically translates and optimizes the task written in LINQ expressions into a Dryad job which is a directed acyclic graph of operators with one process per operator. If possible, connected vertices communicate through shared memory pipes. Otherwise, they communicate through compressed files stored in a distributed file system. The job is then

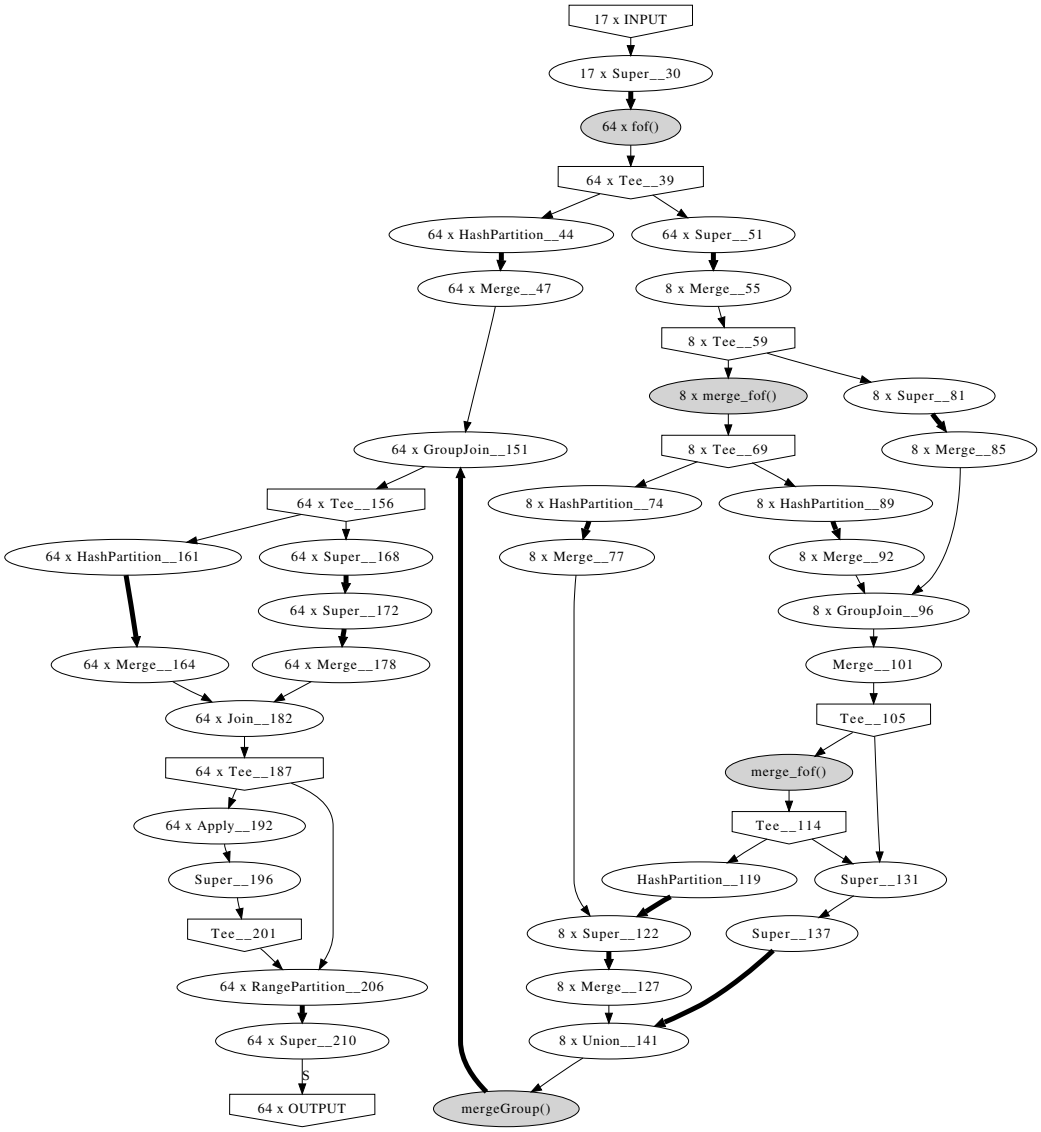


Figure 2.5: **Example Dryad Execution Plan for dFoF.** Nodes in grey color are running user-defined functions. *Partition* and *Merge* vertex pair represents repartitioning of data. *Tee* vertices replicate the input data and feed them to multiple down stream vertices. *Super* vertices are a chain of operators which could be pipelined in memory.

deployed and executed in the Dryad cluster.

We wrote the core `fof()`, `mergefof()` functions as user-defined operators. Because both

functions have to see all data in the input data partition, we used DryadLINQ's `apply` construct which exposes the entire partition to the operator rather than a single data item (*i.e.*, a single particle) at a time. Other than the user defined operators, we used standard LINQ operators not only for the initial data partitioning and relabeling but also for the output post-processing operation of each phase. We also used the lazy evaluation feature of the LINQ framework to implement the iterative hierarchical merge phase. Thus, we only submit a single Dryad job for the entire dFoF task. Using MapReduce, we would have to schedule one MapReduce job for the local clustering, and also one for each iteration of the iterative hierarchical merge process. (See Section 4.3.) The entire job coordination is written in only 120 lines out of a total of 3000 lines. The final Dryad plan to process dataset in Section 2.5 consists of 1227 vertices with three hierarchical merges. An example Dryad execution plan is shown in Figure 2.5.

For the node-local spatial index used in the FoF computation (and also partitioning the data), we chose to use a kd-tree [49] because of its simplicity. We implemented both a standard version of the kd-tree and the optimized version presented in Section 2.3.1.

2.5 Evaluation

In this section, we evaluate the performance and scalability of the dFoF clustering algorithm using two real world datasets. We execute our code on an eight-node cluster running Windows Server 2008 Datacenter edition Service Pack 1. All nodes are connected to the same gigabit ethernet switch. Each node is equipped with dual Intel Xeon E5335 2 GHz quad core CPU, 8 GB RAM, and two 500 GB SATA disks configured as RAID 0. Each Dryad process requires 5 GB RAM to be allocated, or it is terminated. 5 GB per node is just enough to fit 40 GB of the input data in the memory of eight nodes when the input data is evenly distributed among the nodes. So, this memory constraint helps quickly detect unacceptable load imbalances. Note that we tuned neither the hardware nor software configurations other than implementing the algorithmic optimizations that we described previously. Our goal is to show improvements in the relative numbers rather than try and show the best possible absolute numbers.

We evaluate our algorithm on data from a large-scale astronomy simulation that ran on

Percentile	25	50	75	90	99	99.9	100
S43	6	16	97	373	1,853	8,322	10,494
S92	8	44	1,370	41,037	350,140	386,577	387,136
S92:S43	1.33	2.75	14.12	110.02	188.96	46.45	36.89

Table 2.1: Distribution of the number of particles in the two snapshots S43 and S92. Although the two snapshots contain the same number of particles, the distribution of particles in S92 is orders of magnitude denser than that of S43.

2048 compute cores of the Cray XT3 system at the Pittsburgh Supercomputing Center [19]. The simulation itself was only about 20% complete when we acquired the dataset. Therefore we use two relatively early snapshots: S43 and S92 corresponding respectively to 580 million years and 1.24 billion years after the Big Bang. Each snapshot contains 906 million particles occupying 43 GB in uncompressed binary format. Each particle has a unique identifier and 9 to 10 additional attributes such as coordinates, velocity vector, mass, gravitational potential stored as 32-bit floating-point numbers. The data is preloaded into the cluster and is hash partitioned on the particle identifier attribute. Each partition is also compressed in the GZip format. Dryad can directly read compressed data and decompresses it on-the-fly as needed.

For this particular simulation, astronomers set the distance threshold (ϵ) to 0.000260417 in units where the size of the simulation volume is normalized to 1. Both datasets require at least two levels of hierarchical merging.

As the simulation progresses, the Universe becomes increasingly structured (*i.e.*, more stars and galaxies are created over time). Thus, S92 has not only more clusters (3496) than S43 (890) but also has denser regions than S43. Table 2.1 shows the distribution of the number of particles within distance threshold (*i.e.*, density of data).

Ideally, the structure of data should not affect the runtime of the algorithm so that scientists can examine and explore snapshots taken at any time of simulation in a similar

amount of time.

Evaluation summary. In the following subsections, we evaluate our dFoF Dryad implementation. First, we process both snapshots using an eight-node Dryad cluster while varying the partitioning scheme and the spatial index implementation. These experiments enable us to measure the overall performance of the algorithm and the impact of our two optimizations. Second, we evaluate dFoF's scalability by varying the number of nodes in the cluster and the size of the input data. Finally, we compare dFoF to the existing OpenMP implementation that the astronomers use today. Overall, we find that dFoF exhibits a near linear speedup and scaleup even with suboptimal hardware and software configurations. Additionally, dFoF shows consistent performance regardless of skew in the input data thanks to the optimization in Section 2.3.

2.5.1 Performance

In this section, we use the full eight-node cluster, varying the partitioning scheme and spatial index implementation. For the partitioning scheme, we compare deterministic uniform partitioning (Uniform) described in Section 2.2 and dynamic partitioning (Non-uniform) described in Section 2.3.2. We also compare an ordinary kd-tree implementation (Normal) to the optimized version (OPT) described in Section 2.3.1. We repeat all experiments three times except for Uniform partitioning using the Normal kd-tree because it took over 20 hours to complete. For Non-uniform partitioning, we use a sample of size 0.1%. We show the total runtime including sampling and planning times. There is no special reason for using a small sample except to avoid a high overhead of planning. As the results in this section show, even small samples work well for this particular dataset.

Figure 2.6 shows the total run times for each variant of the algorithm and each dataset. For dataset S43, which has less skew in the cluster-size distribution, all variants complete within 70 minutes. However, when there is high skew (*i.e.*, more structures as in S92), the normal kd-tree implementation takes over 20 hours to complete while the optimized version still runs in 70 minutes. Uniform-OPT over snapshot S92 did not complete because

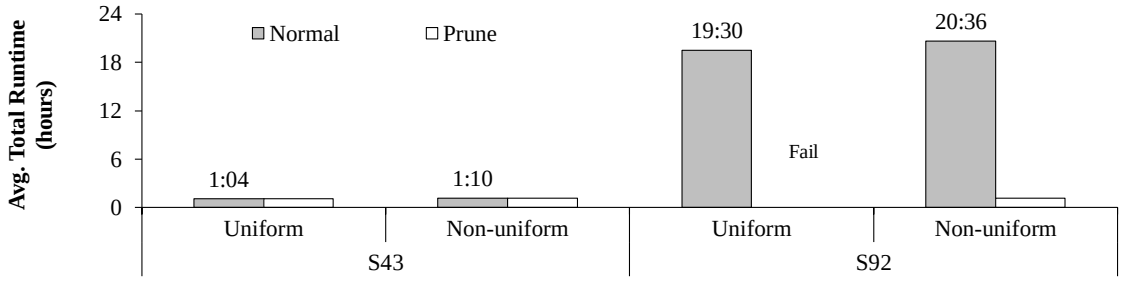


Figure 2.6: **Average time to cluster entire snapshot.** Average of three executions except for jobs that took longer than 20 hours. Missing bar is due to a failure caused by an out-of-memory error due to slightly larger memory footprint of the optimized index than the default kd-tree (see Figure 2.9). As the figure shows, with both non-uniform partitioning and optimized index enabled, both snapshots are processed within 70 min.

it reached full memory capacity while processing a specific data partition, causing the failure of the entire query plan as we discuss in more detail below.

Figure 2.7 shows the average relative time taken by each phase of the algorithm. The hierarchical merge occurs in order of mergeLv1, mergeLv2, and mergeLv3. As the figure shows, local clustering, `fof`, takes more than 40% of total runtime in all cases and completely dominates when there is high-skew in the data and a normal kd-tree is used. All other user-defined functions account for less than 4% of total runtime. All other standard operators account for over 50% of total runtime, but the total is the sum of more than 30 operators including repartitions, filters, and joins to produce intermediate and final result for each level of the hierarchical merge. In the following subsections, we report results only for the dominant `fof` phase of the computation and analyze the impact of different partitioning schemes and different spatial index implementations.

In Figures 2.8 and 2.9, we measure the *per-node* runtime and peak memory utilization of the local `fof` phase. We plot the quartiles and minimum and maximum values. Low variance in runtime represents a balanced computational load, and low variance in peak memory represents balance in both computation and data across different partitions. In both Figures 2.8 and 2.9, Non-uniform partitioning shows a tighter distribution in runtime

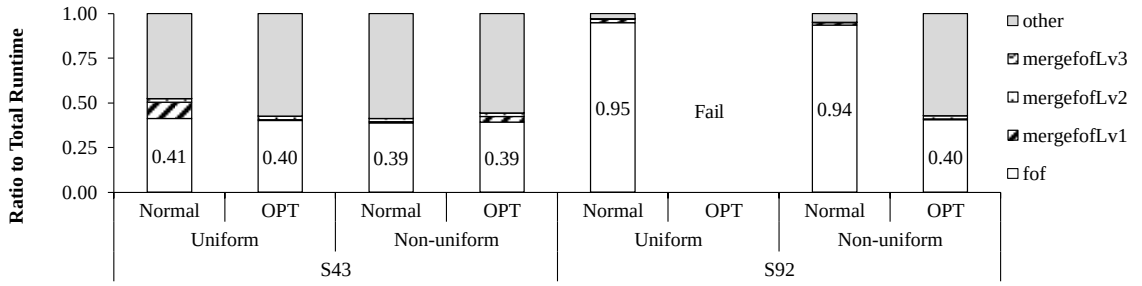


Figure 2.7: **Runtime breakdown across phases.** Average of three executions except jobs that take longer than 20 hours. The initial `fof()` takes more than 40% of total runtime. The three-level hierarchical merges, `mergefof()`, took less than 4% of total runtime. "Other" represents time to take to run all standard vertices such as filter, partition, joins to glue the phases. Overall, `fof()` is the bottleneck and completely dominates when the data is highly skewed and an ordinary kd-tree is used.

and peak memory utilization than uniform partitioning. With uniform partitioning, the worst scenario happens when we try the optimized kd-tree implementation. Due to high data skew, one of the partitions runs out of memory causing the entire query plan to fail. This does not happen with normal kd-tree and uniform partitioning because the optimized kd-tree has a larger memory footprint as discussed in Section 2.3.1. Non-uniform partitioning is therefore worth the extra scan over the entire dataset.

As Figure 2.8 shows, dFoF with the optimized index (Section 2.3.1) significantly outperforms Normal implementation especially when there is significant skew in the cluster-size distribution. Thanks to the pruning of visited subtrees, the runtime for S92 remains almost the same as that for S43. However, the optimization is not free. Due to the extra tracking flag, the optimization requires slightly more memory than the ordinary implementation as shown in Figure 2.9. The higher memory requirement could be alleviated by a more efficient implementation such as keeping a separate bit vector indexed per node identifier or implicitly constructing a kd-tree on top of an array rather than keeping pointers to children in each node. Overall, however, the added memory overhead is negligible compared with

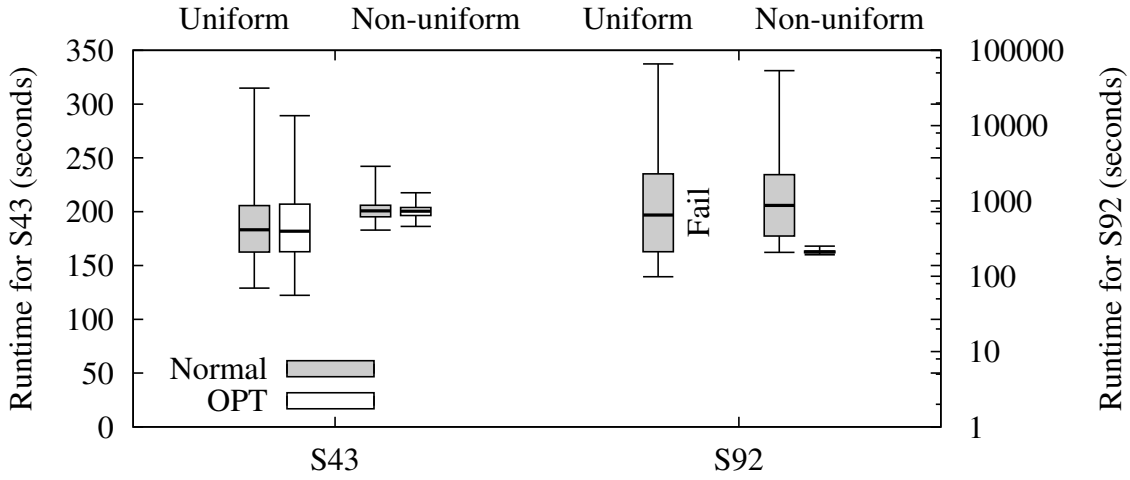


Figure 2.8: **Distribution of FoF runtime per partition.** Uniform partitioning yields higher variance in per partition runtime than Non-uniform partitioning. FoF with optimized index traversal runs orders of magnitude faster than FoF with normal implementation in S92 dataset. Note that the y-axis for S92 is in log scale.

the order-of-magnitude gains in runtime.

2.5.2 Scalability

In this section, we evaluate the scalability of the dFoF algorithm with non-uniform data partitioning and the optimized kd-tree. In these experiments, we vary the number of nodes in the cluster and redistribute the input data only to the participating nodes. All reported results are the average of three runs. The standard deviation is less than 1%.

Figure 2.10 shows the runtime of dFoF for each dataset and increasing number of compute nodes. *Speedup* measures how much faster a system can process the same dataset if it is allocated more nodes [41]. Ideally, speedup should be linear. That is, a cluster with N nodes should process the same input data N times faster than a single node. For both datasets, the runtime of the dFoF is approximately half as long as we double the number of nodes, showing a close-to perfect linear speedup. We do not present the number for the

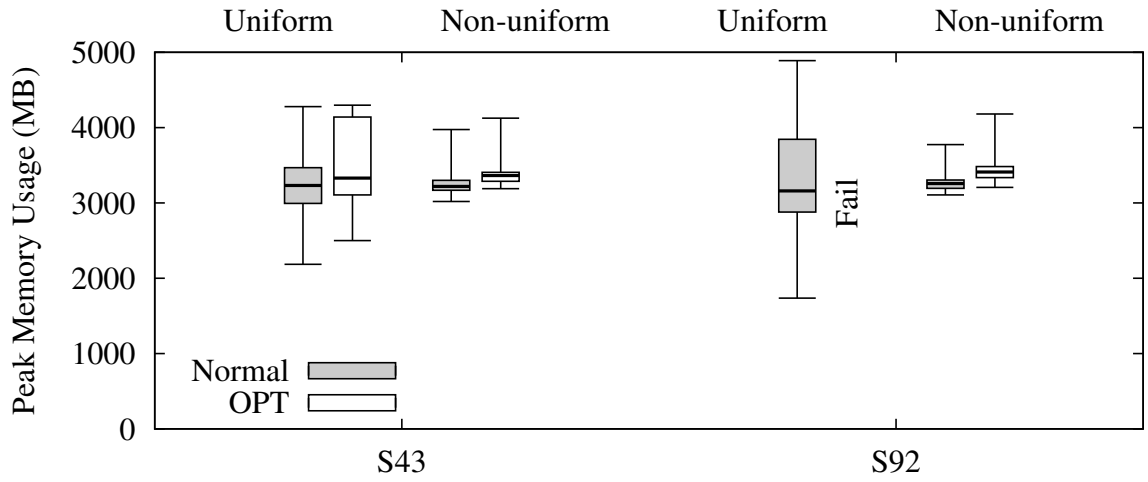


Figure 2.9: **Distribution of FoF peak memory utilization per partition.** Uniform partitioning yields a higher variance in peak memory utilization per partition than Non-uniform partitioning. This high variance caused the crash of one of the partitions with optimized index traversal. The optimized kd-tree index traversal has a higher memory footprint than the normal implementation.

single-node case due to an unknown problem in the Dryad version we use; the system did not schedule remaining operators if currently running operator takes too long to complete.

Figure 2.11 shows the scaleup results. Scaleup measures how a system handles data size that has increased in proportion to the cluster size. Ideally, as the data and cluster size increase proportionally to each other, the runtime should remain constant. To vary the data size, we subsample the S43 and S92 datasets. For 4-node and 8-node configurations, the scaleup is close to ideal: the ratio of runtimes to the single-node case are 0.99 and 0.91 respectively. The 2-node experiment showed a scaleup of only 0.83 and 0.78. We investigated the 2-node case and found that the size of the subsampled dataset was near the borderline of requiring one additional hierarchical merge. Thus, each partition was underloaded and completed quickly, overloading the job scheduler and yielding a poorer scaleup.

Overall, considering our suboptimal hardware configuration, the scalability of dFoF is

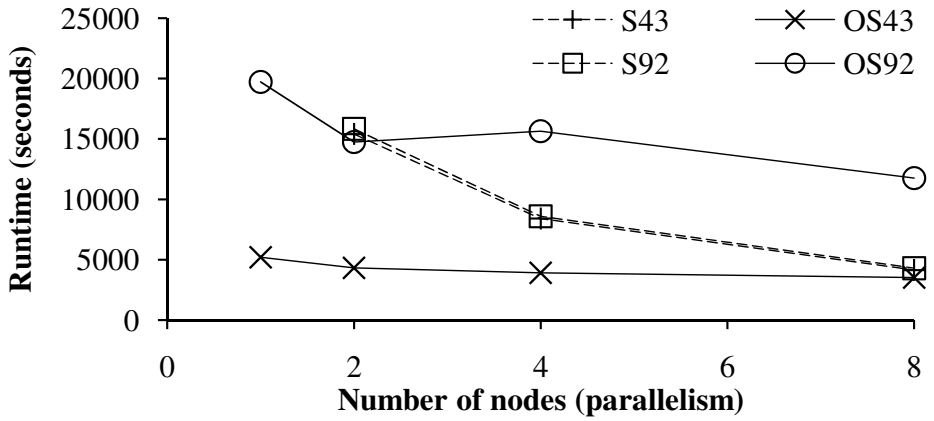


Figure 2.10: **Speedup.** dFoF runtime for each dataset with varying number of nodes. dFoF speedup is almost linear. OS43 and OS49 are the result of OpenMP implementation of FoF with varying degree of parallelism. Note that S43 overlaps with S92.

excellent.

2.5.3 Compared to OpenMP implementation

Astronomers currently use a serial FoF implementation that has been moderately parallelized using OpenMP [32] as a means of scheduling computation across multiple threads that all share the same address space. OpenMP is often used to parallelize programs that were originally written serially. The two biggest drawbacks of OpenMP are (1) non-trivial serial portions of code are likely to remain, thereby limiting scalability by Amdahl's Law; (2) the target platform must be shared memory. The serial aspects of this program are state-of-the-art in terms of performance — they represent an existing program that has been performance-tuned by astrophysicists for over 15 years. It uses an efficient kd-tree implementation to perform spatial searches, as well as numerous other performance enhancements. The OpenMP aspects are not performance-oriented, though. They represent a quick-and-dirty way of attempting to use multiple processing cores that happen to be present on a machine with enough RAM to hold a single snapshot.

The shared-nothing cluster that we used for the previous experiments represents a com-

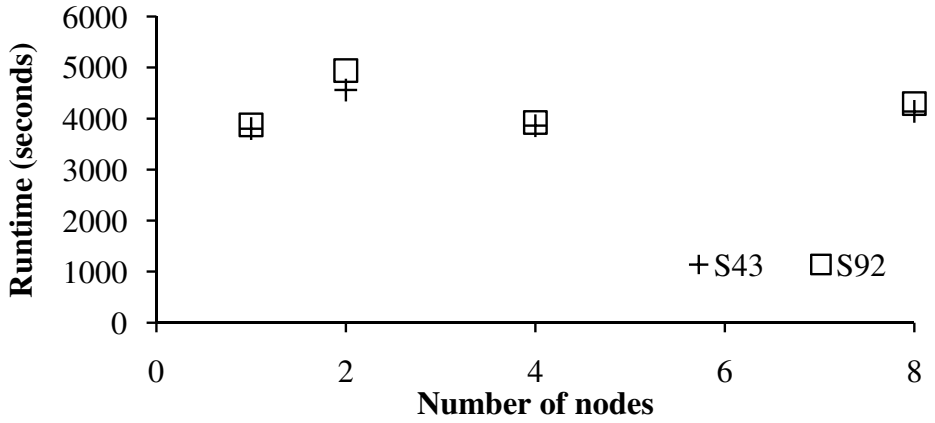


Figure 2.11: **Scaleup.** Runtime of dFoF with increasing data size proportional to the number of nodes. Except for the two node case where scheduling overhead dominated, dFoF scales up in linear.

mon cost-efficient configuration for modern hardware: roughly 8 cores per node and one to two GB of RAM per core. Our test dataset is deliberately much larger than what can be held in RAM of a single one of these nodes. The astrophysics FoF application must therefore be run on an unusually large shared-memory platform. In our case, the University of Washington Department of Astronomy owns a large shared-memory system with 128 GB of RAM, 16 Opteron 880 single-core processors running at 2.4 GHz, and 3.1 TB of RAID 6 SATA disks.

At the scale of 128 GB of RAM, it is now cheaper to buy a single shared-memory system than to distribute the same 128 GB across 16 nodes. However, this cost-savings breaks down at the scales beyond 128 GB. For example, it is not possible to find symmetric multiprocessing systems (SMPs) with 1TB of RAM. At this scale, certain vendors offer systems with physically distributed memory that share a global address space ("Non-Uniform Memory Access" or "NUMA" systems), but these are generally more expensive than building a cluster of distributed-memory nodes from commodity hardware. Furthermore, the ostensible advantage of the shared-nothing architecture over a large, shared-memory system is the scalability of I/O.

Consequently, our goal is to achieve competitive performance with the astrophysics FoF running on the shared-memory system with our Dryad version running on the shared-nothing cluster (*i.e.*, 64 GB of total RAM — just barely large enough to fit the problem in memory). If we do this, then we have demonstrated that the MapReduce paradigm is an effective means of utilizing cheaper distributed-memory platforms for clustering calculations at scales large enough to have economic impact.

In order to normalize serial performance, we ran the existing astrophysics FoF application on a smaller dataset on both the shared-memory system and our cluster. The dataset was small enough to fit completely into RAM on a single cluster node. The shared-memory platform took 61.4 seconds to perform the same analysis that required 34.8 seconds on a cluster node excluding I/O. We do not include I/O in our normalization because the system's storage hardware is still representative of the current state-of-the-art; only its CPUs are dated. In the following results, we normalize the timings of the CPU portion of the test on shared-memory system to the standard of the Dryad cluster hardware.

Running the astronomy FoF algorithm in serial on the shared-memory system for our test dataset S43 (with the same parameters as our cluster runs) took 5202 seconds in total — only 1986 of this was actual FoF calculation, the rest was I/O. In comparison, our Dryad version would likely have taken an estimated 30,000 seconds, as extrapolated from our optimized Dryad 2-node run assuming ideal scalability. However, since we do not actually know the serial runtime of Dryad on this dataset, it is difficult to compare a parallel Dryad run directly to our serial FoF implementation, since there is undoubtedly parallel overhead induced by running Dryad on more than one node.

The runtime comparisons are much more interesting for S92. The particle distribution in S92 is more highly clustered than S43, meaning that the clusters are larger on average, and there are more of them. In this case, the astrophysics FoF takes quite a bit longer: 16763 seconds for the FoF computation itself and 19721 for the entire run including I/O, compared to roughly 30,000 seconds for a serial Dryad run of the same snapshot. The OpenMP implementation still wins, but the difference is smaller than for S43.

One can also see the effect of S92's higher clustering on the OpenMP scalability. The OpenMP version is not efficient for snapshots with many groups spanning multiple thread

domains. This limitation is because multiple threads may start tracking the same group. When two threads realize they are actually tracking the same group, one gives up entirely but does not contribute its already-completed work to the survivor. While this is another optimization that could be implemented in the OpenMP version, astronomers have not yet done so. This effect can be seen in Figure 2.10.

Since our Dryad version performed similarly on both snapshots, we conclude that our methodology achieves scalability in both computational work and I/O. The advantage of our implementation can be seen when we run on more nodes. This advantage allows us to match the performance of the astrophysics code on S43 (3513 seconds vs. 4141 seconds) and to substantially outperform it for S92 (11763 seconds vs. 4293 seconds). This idea is in keeping with the MapReduce strategy: We employ a technique that may be less than optimally efficient in serial, but that scales very well. Consequently, we have achieved our goal of reducing time-to-solution on platforms that offer an economic advantage over current shared-memory approaches at large scales.

2.6 Conclusion

Science is rapidly becoming a data management problem. Scaling existing data analysis techniques is very important to expedite the knowledge discovery process. In this chapter, we designed and implemented a standard clustering algorithm to analyze astrophysical simulation output using a popular MapReduce-style data analysis platform. Through experiments on two real datasets and a small eight-node lab-size cluster, we show that our proposed dFoF algorithm achieves near-linear scalability and performs consistently regardless of data skew. To achieve such performance, we leverage non-uniform data partitioning based on sampling and introduce an optimized spatial index approach. An interesting area of future work, is to extend dFoF to other density-based clustering algorithm such as the DBSCAN algorithm [46] or the OPTICS algorithm [8].

Chapter 3

STUDY OF SKEW IN MAPREDUCE APPLICATIONS

In Chapter 2, we showed how a naïve implementation of a UDO can cause significant performance degradation due to skew. In this chapter, we study the skew problem in several other MapReduce applications and analyze three cluster workloads to measure the prevalence of this problem. More specifically, we first present an overview of the MapReduce programming model and show how a MapReduce job is structured and executed in Hadoop, an open source MapReduce engine (Section 3.1). We then analyze five different causes of the skew problem in several real MapReduce applications (Section 3.2). We also analyze workloads of three research Hadoop clusters and examine the severity of the skew problem in practice and the effectiveness of speculative execution (the only countermeasure implemented in MapReduce and Hadoop to handle load imbalance among tasks) for mitigating this problem (Section 3.3).

Overall, we find that more than 40% of jobs running more than five minutes have at least one task running at least 50% longer than the median runtime of its peer tasks. More than half of such long-running tasks are running at least twice and up to orders of magnitude longer than the median runtime of their peer tasks. We also find that only 20% of speculative execution attempts are successful (*i.e.*, one of the speculated tasks completes faster than the initial attempt). We conclude the chapter with five best practices in authoring MapReduce applications (Section 3.4).

3.1 MapReduce Programming Model

Dean and Ghemawat introduced the MapReduce API and runtime system to write and execute distributed data processing applications in a fleet of commodity PC servers [37]. The original MapReduce paper subsequently inspired an open source project Hadoop [61]. The Hadoop project closely replicates the MapReduce system including the distributed file

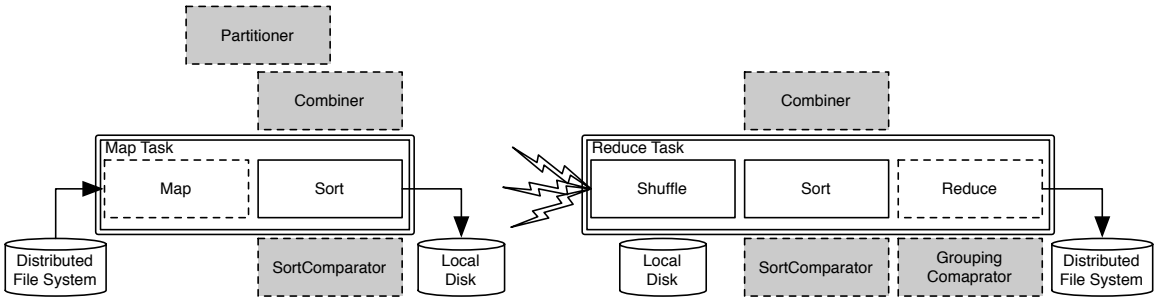


Figure 3.1: An execution of a MapReduce job in Hadoop. Dashed boxes represent user-supplied functions. Shaded boxes represent optional functions. White boxes define phases in a MapReduce job execution. Data flows from left to right in the figure.

system (*i.e.*, Hadoop Distributed File System (HDFS) [23] instead of the Google File System (GFS) [54]) described in the original MapReduce paper.

With the MapReduce programming model, a user can easily write a distributed data processing application. The programming model requires two functions: map and reduce.

$$map : (K1, V1) \rightarrow [(K2, V2)]$$
$$reduce : (K2, [V2]) \rightarrow [(K3, V3)]$$

The map function is invoked per input record (*i.e.*, per key-value pair) of any key type K1 and any value type V1 and produces a bag of output records with key type K2 and value type V2. The map output records are then grouped by the output key, and all values that have the same key are passed to the reduce function. The reduce function is thus invoked per map output key, or reduce key, and produces final output records. For given map and reduce functions, the framework handles all runtime issues such as parallelization, scheduling, and fault-tolerance.

Figure 3.1 shows the structure of a typical MapReduce job in Hadoop. A MapReduce job usually reads its input from and writes its output to the underlying distributed file system. The distributed file system stores a file as a sequence of fixed-size blocks and each block is replicated for fault-tolerance. A MapReduce *job* consists of two types of tasks: *map tasks*

and *reduce tasks*. Each task executes in phases (white boxes in Figure 3.1). By default, a map task is created for each block of the input file. Each map task applies the given map function to all assigned input records (Map). The output of the map function is typically hash-partitioned and sorted by the reduce key (Local Sort). The number of reduce tasks is typically chosen by the user. When a map task completes, the reduce tasks are notified to pull newly available data (Shuffle). Once all map tasks complete and all reduce input is shuffled, the reduce input is sorted on the reduce key (Sort) then the reduce tasks start the reduce phase. They invoke the reduce function once for each reduce key. Each invocation processes one key and all associated values (Reduce). The final output is stored back in the distributed file system.

There is a coordinator daemon called *JobTracker* for each Hadoop MapReduce cluster. The JobTracker manages the life cycle of each MapReduce job and makes scheduling decisions across jobs and across tasks. A *TaskTracker* daemon runs at each worker node in the cluster. The TaskTracker manages the life cycle of each task executing on the node. The TaskTracker periodically contacts the JobTracker and reports the current status and progress of the tasks that it manages. It also retrieves a new task to run from the JobTracker if it has the capacity to run that task. A cluster configuration parameter determines how many map or reduce tasks a node can run simultaneously. This setting is referred to as the number of *slots*.

In the Hadoop implementation of MapReduce, users can further control their applications by providing optional functions (shaded boxes in Figure 3.1).

$$\begin{aligned} partitioner &: (K2, V2) \rightarrow int \rightarrow int \\ sortComparator &: K2 \rightarrow K2 \rightarrow int \\ groupingComparator &: K2 \rightarrow K2 \rightarrow int \\ combiner &: (K2, [V2]) \rightarrow [(K2, V2)] \end{aligned}$$

With *partitioner*, users can override the default hash-partitioning of the map output and use more sophisticated partitioning schemes (*e.g.*, range partitioning, partitioning using a space-filling curve). The partitioner is invoked once for each map output. With *sortComparator*, the users can control the order of the reduce input. sortComparator is useful when a user

wants to sort the final output in a specific order. With *groupingComparator* users can group reduce keys and invoke the reduce function per group rather than per reduce key. groupingComparator is a popular way to implement a reduce-side join [20]. The *Combiner* performs a partial aggregation over a small batch of map output to reduce the data size transferred over the network. Similarly to the reduce function, the combiner is invoked once for each reduce key and processes all the map output that shares the same reduce key in the batch.

3.2 Types of Skew in a MapReduce Application

We present five types of skew that can arise in a MapReduce application. We only consider skew originating from the characteristics of the algorithm and dataset. We do not consider load imbalance due to workload interference and heterogeneous resources. For each type of skew, we relate a real world application where we encountered this source of skew in practice.

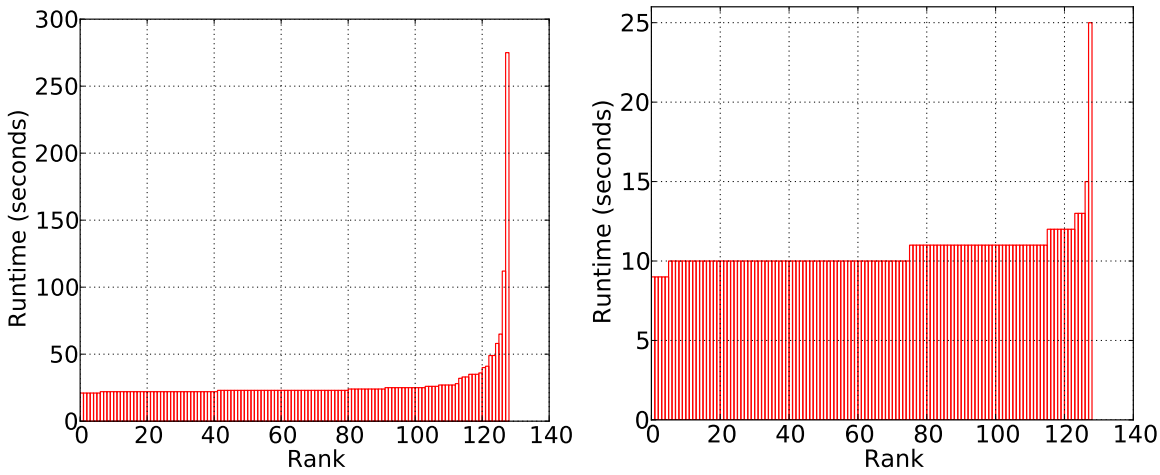
3.2.1 Sources of Map-side Skew

We identify three causes of skew in the map phase.

Expensive Record

Map tasks typically process a collection of records in the form of key-value pairs, one-by-one. Ideally, the processing time does not vary significantly from record to record. However, depending on the application, some records may require more CPU and memory to process than others. These expensive records may simply be larger than other records, or the map algorithm's runtime may depend on the record's value.

PageRank [25] is an application that can experience this type of skew. PageRank is a link analysis algorithm that assigns weights (ranks) to each vertex in a graph by iteratively aggregating the weights of its inbound neighbors. This application can thus exhibit skew if the graph includes nodes with a large degree of incoming edges. We took the PageRank implementation from Cloud 9 [72] and applied it to the freebase dataset [55]. We patched



(a) Page Rank - Map

(b) Page Rank - Reduce

Figure 3.2: The distribution of task runtimes for PageRank with 128 map and 128 reduce tasks. A graph node with a large number of edges is much more expensive to process than many graph nodes with few edges. Skew arises in both the map and reduce phases, but the overall job runtime is dominated by the map phase.

Cloud 9 so that it would properly handle graph nodes with large numbers of edges without running out of memory. The freebase graph is 2 GB in size, and contains 37M nodes and 342M edges. We stored the graph in a Hadoop sequence file, hash-partitioned on node id[1].

Cloud 9 expresses each iteration of PageRank as a sequence of two MapReduce jobs. The first MapReduce job distributes the weight of each vertex in the graph along outgoing edges during the map phase, then aggregates these weights into a new PageRank value for each vertex during the reduce phase. The second MapReduce job handles the random jump and lost weights due to dangling vertices in the first MapReduce job. The output of the second MapReduce job is the newly weighted graph. This graph forms the input to the next iteration. We observed skew in the first job. Figure 3.2a shows the distribution of map task

[1]Cloud 9 provides multiple ways to partition data and multiple implementations of the PageRank algorithm. We chose the most straight forward schemes: hash partition and best practice implementation with combiner.

runtimes in that job, during the first iteration of the algorithm (subsequent iterations show similar trends). The total runtime of this job is approximately 5 minutes. In Figure 3.2a, the longest map task takes more than four minutes while most map tasks complete in 30 seconds. After investigation, we found that the slow map tasks were processing graph nodes with a large number of outgoing edges. These graph nodes were significantly slower to process, leading to the skew shown in the figure.

Heterogeneous Maps

MapReduce is a unary operator, but can be used to emulate an n-ary operation by logically concatenating multiple datasets into a single input file. Each dataset may be required to be processed differently, leading to a multi-modal distribution of task runtimes.

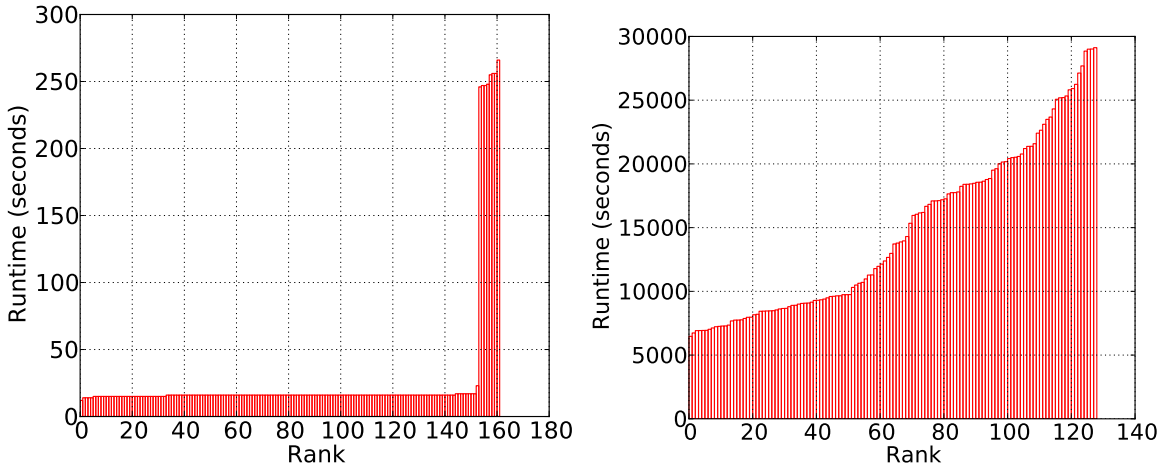
For example, SkewedJoin is one of the join implementations in the Pig system [52]. Each map task in SkewedJoin distributes frequent join keys from one of the input datasets in a round-robin fashion to reduce tasks, but broadcasts joining records from the other dataset to all reduce tasks. These two algorithms exhibit different runtimes because the map tasks that perform the broadcasts do more I/O than the other map tasks.

CloudBurst [116] is a MapReduce implementation of the RMAP algorithm for short-read gene alignment[2]. CloudBurst aligns a set of genome sequence reads with a reference sequence. CloudBurst distributes the approximate-alignment computations across reduce tasks by partitioning the reads and references on their n-grams. The references and reads bearing frequent n-grams are handled similarly to frequent join keys in SkewedJoin: frequent n-grams from a reference sequence are replicated, and frequent n-grams from a read are distributed in round-robin.

Figure 3.3a shows the runtime distribution of map tasks in the CloudBurst application.[3] The total runtime for the job is over 8 hours. Unlike the PageRank application, the runtime

[2] `http://rulai.cshl.edu/rmap/`

[3] We ran the CloudBurst job on a biology dataset [73]. For each alignment, we allowed up to 4 mismatches including insertion and deletion. We used 160 map tasks and 128 reduce tasks for the entire alignments. We use 64 reduce tasks to process low-complexity fragments. The reduce phase processes 128 sequences at a time (first loading data from reference dataset in memory, then processing 128 sequences from the query dataset in a batch).



(a) CloudBurst - Map

(b) CloudBurst - Reduce

Figure 3.3: Distribution of task runtime for CloudBurst. Total 162 map tasks, and 128 reduce tasks. The map phase exhibits a bimodal distribution. Each mode corresponds to map tasks processing a different input dataset. The reduce is computationally expensive and has a smooth runtime distribution, but there is a factor of five difference in runtime between the fastest and the slowest reduce tasks.

distribution during the map phase exhibits a bimodal distribution and there is little variance within each mode. We verified that the two modes correspond to the two input datasets. Although there is no significant skew within each mode, the MapReduce job is experiencing skew because the two modes coexist in a single job.

Non-Homomorphic Map

One of the key features of the MapReduce framework is that users can run arbitrary code as long as it conforms to the MapReduce interface: map() or reduce(), and typically initialization and cleanup. Such flexibility enables users to push, when necessary, the boundaries of what map and reduce phases have been designed to do: each map output can depend on a group of input records — *i.e.*, the map task is non-homomorphic. For example, although the conventional join algorithm in MapReduce requires both map and reduce phases, if the data

are sorted on the join attribute, the join can be implemented directly in the map phase using a sort-merge algorithm. Similarly, a clustering algorithm can directly run during the map phase if the data are already partitioned by a prior MapReduce job [85]. In these scenarios, a map task may run what is normally reduce logic such as aggregation or join, consuming a group of records as a unit rather than a single record as in a typical MapReduce application. Thus, the map tasks may experience reduce-side skews discussed in Section 3.2.2.

Users can also employ the MapReduce framework to implement a distributed analysis application by only leveraging the distributed execution and fault-tolerance features of the MapReduce engine (*e.g.*, [81]). In such scenarios, the map phase often runs arbitrary computation which is potentially non-homomorphic[4].

In the above scenarios, map tasks may run a CPU-intensive algorithm over many input records. If the runtime of the algorithm varies depending on the distribution of input data or the relationships between input data, then a job may incur significant map-skew.

An example of such an application is a data clustering algorithm called Friends of Friends (FoF) [33] that we have implemented in multiple MapReduce-type systems, including Hadoop (we present the dFoF implementation in detail in Chapter 2 and 4). FoF is used by astronomers to analyze the structure of the universe within a snapshot of a simulation of the universe evolution. For each point in the dataset, the FoF algorithm uses a spatial index to recursively look up neighboring points to find connected clusters. The performance of a range query over a spatial index varies depending on the data distribution. In a dense region, every lookup returns a large number of neighbors, but in a sparse region the lookup returns few records. Thus, processing times depend on the distribution of input data to map tasks.

Figure 3.4 shows the runtime distribution of the FoF "local clustering phase" [81, 85], which runs in the map tasks. The data is space-partitioned by a prior MapReduce job. There are 276 map tasks. Each task is assigned a region of space such that all tasks have the same amount of data in bytes and in number of records (they differ by less than 2%). Even with this condition enforced, the runtime varies between 6 minutes and 13 hours.

[4]The new interface since Hadoop 0.20 makes writing this kind of applications easier and cleaner.

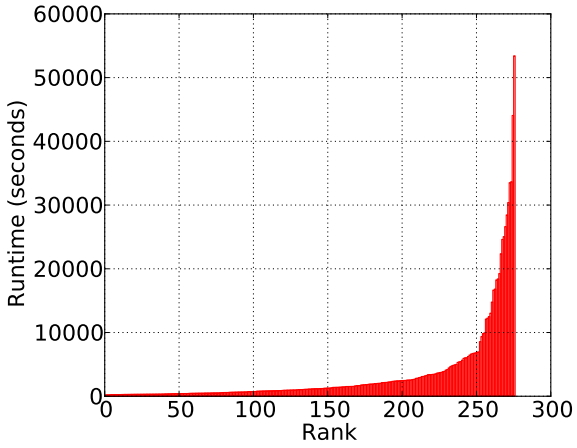


Figure 3.4: Runtime distribution of the local clustering phase of the Friends-of-Friends algorithm [81, 85]. Total 276 map tasks. Even though all map tasks received the same amount of data, the slowest map takes more than 50000 seconds while the fastest one completes in 400 seconds due to different input data value distributions.

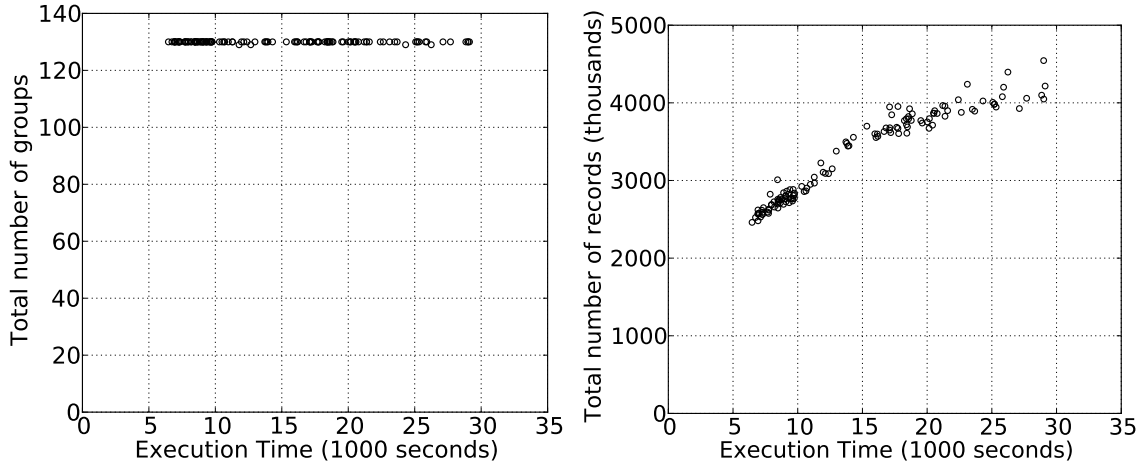
3.2.2 Reduce-side skew

We identify two types of reduce-skew. The first one, *partitioning skew*, is unique to reduce. The other type of skew is analogous to the map-skew problems above.

Partitioning skew

In MapReduce, the outputs of map tasks are distributed among reduce tasks via hash-partitioning (by default) or some user-defined partitioning logic. The default hash-partitioning is usually adequate to evenly distribute the data. However, reduce-skew can still arise in practice. Consider the following two examples.

First, consider an application that needs to process many small files. In Hadoop, processing a small number of large files is more efficient than processing a large number of small files. As a result, users often write MapReduce jobs that combine small files into larger *sequence files*. One of our science collaborators wrote such a MapReduce job. The map derives the target sequence file name from the content of the small files, then the out-



(a) Number of key groups (b) Number of input records

Figure 3.5: Distribution of the number of key groups and input records per reduce task with respect to runtime for CloudBurst. Each point in the figures represents a reduce task. The number of key groups assigned to each reducer is 129 or 130 key groups. It is thus extremely well balanced. However, there is factor of five difference between the minimum and the maximum task runtime. This suggests that evenly distributing key groups among reduce tasks is not sufficient to eliminate the skew for CloudBurst. The number of input records assigned to each reduce task varies by a factor of two and the runtime increases as the number of records increases. This suggests that the number of input records is a better measure to partition the map output than the number of key groups for CloudBurst.

put is distributed among reduce tasks by hash-partitioning on the target sequence file name with a default hash function. The user wanted to assign one reduce task for each sequence file but unfortunately the hash partitioning scheme did not evenly distribute the key groups (*i.e.*, file names) across the available reduce tasks because there are too few key groups compared to the number of reduce tasks. This problem is known as the coupon collector's problem [34]. Roughly $n \log n$ distinct key groups are necessary to prevent an empty reduce task (*i.e.*, a reduce task with no input data). In this scenario, there exist only n key groups thus it is likely that some reduce tasks are assigned more than one key group, leading to

skew. Therefore, some reduce tasks ended up writing multiple sequence files, each in the order of a TB, while others completed almost immediately.

As a second example, even when the partitioning function perfectly distributes keys across reducers, some reducers may still be assigned more data simply because the key groups they are assigned contain significantly more values. Figure 3.3b shows the runtime distribution with respect to the number of input key groups and the number of input records per reduce task for the CloudBurst application above. While the keys are distributed evenly across the reduce tasks (Figure 3.5a), there is a factor of two difference between the smallest and the largest key-group in the number of input records (Figure 3.5b). As a result, the runtime of the reducers exhibits skew.

In general, balanced data allocation is a difficult problem if the partitioning logic relies upon values computed during the execution of the map algorithm because the values are not known beforehand.

Expensive Input

In MapReduce, reduce tasks process a sequence of (key, set of values) pairs. As in the case of expensive records processed by map, expensive (key, set of values) pairs can skew the runtime of reduce tasks. Since reduce operates on key groups instead of individual records, the expensive input problem can be more pronounced, especially when the reduce is a holistic operation that requires memory proportional to the size of the input data [57]. A *holistic reduce* may load the entire associated values with a reduce key in memory and run complex algorithms (*e.g.*, find clusters in a multi-dimensional input data using a spatial index, perform complex joins, and analyze the activities of a user given a subgraph of social network).

The runtime of a holistic reduce that runs a complex algorithm can significantly vary per reduce key. For example, the reduce of CloudBurst performs a similarity join. It is a holistic reduce because each reduce invocation needs to buffer all the values from the reference dataset. Figure 3.5a shows that all reduce tasks are extremely well balanced in terms of the reduce keys. However, there is factor of five difference between the minimum

and the maximum task runtime. This suggests that some reduce keys are more expensive to process than others. As shown in Figure 3.5b, there is a factor of two difference in the number of input records between the fastest and the slowest reduce tasks, but there is a factor of five difference in runtime between tasks. Thus, each reduce in the slowest reduce task processed 2x more input records and each input record is 2.5x more expensive than in the fastest reduce task on average. The difference might have been greater depending on the data and the parameters of the algorithms.

3.3 Skew in the Real World

How frequent is the skew problem in practice? How significant is the impact of the skew problem? To answer these questions, we analyze the workloads of three research Hadoop clusters. As shown in the previous section, the tasks that exhibit skew run significantly longer than other tasks in the same job. We call such long running tasks *straggler tasks*.

In this section, we first analyze the frequency and distribution of straggler tasks (Section 3.3.2) then analyze the relationship between skew and techniques to partition input data (Section 3.3.3). Finally, we evaluate the effectiveness of speculative execution in Hadoop, which is the default mechanism to address straggler tasks (Section 3.3.4).

3.3.1 Datasets and Scope of Analysis

We analyze the execution logs from three Hadoop MapReduce clusters used for research: OpenCloud, M45, and Web Mining. The three clusters have different hardware and software configurations and range in size from nine nodes (Web Mining), to 64 nodes (OpenCloud), to 400 nodes (M45).

OpenCloud: OpenCloud is a research cluster at Carnegie Mellon University (CMU) managed by the CMU Parallel Data Lab. It is open to researchers (including faculty, post-docs, and graduate students) from all departments on campus. In the trace that we collected, the cluster was used by groups in areas that include computational astrophysics, computational biology, computational neurolinguistics, information retrieval and information classification, machine learning from the contents of the Web, natural language

Cluster	Duration	Start Date	End Date	Successful/Failed/Killed Jobs	Users
OpenCloud	20 months	2010 May	2011 Dec	51975/4614/1762	78
M45	9 months	2009 Jan	2009 Oct	42825/462/138	30
Web Mining	5 months	2011 Nov	2012 Apr	822/196/211	5

Table 3.1: Summary of Analyzed Workloads

processing, image and video analysis, security malware analysis, social networking analysis, cloud computing systems development, cluster failure diagnosis, and several class projects related to information retrieval, data mining and distributed systems.

The 64 nodes in this cluster each have a 2.8 GHz dual quad core CPU, 16GB RAM, 10 Gbps Ethernet NIC, and four Seagate 7200 RPM SATA disk drives. The cluster ran Hadoop 0.20.1 during the data collection.

M45: M45 is a production cluster made available by Yahoo! to support scientific research [1]. The research groups that used this cluster during the collection of the trace covered areas that include large-scale graph mining, text and web mining, large-scale computer graphics, natural language processing, machine translation problems, and data-intensive file system applications [75]. The 400 nodes in this cluster each contain two quad-core 1.86GHz Xeon processors, 6GB of memory, and four 7200 rpm SATA 750 GB disk. The cluster ran Hadoop 0.18 with Pig during the data collection. The first four months of the trace overlap with the end of the trace previously analyzed by Kavulya *et al.* [75].

Web Mining: Web Mining is a small cluster owned and administered by an anonymized research group. The group comprises faculty, post-docs, and students. Most users run web text mining jobs. The 9 nodes in the cluster each have four quad-core CPU Xeon E5630, 32GB RAM, four 1.8TB HDD. The cluster runs Hadoop 0.20.203 with Pig.

Log: Table 3.1 summarizes the duration of each collected log and the number of Hadoop jobs that it covers. For each cluster, our log comprises two types of information for each executed job. All data were collected automatically by standard Hadoop tools requiring no additional tracing tools.

Cluster	Total	> 5 mins	$\|map\| > 1$	$\|reduce\| > 1$
OpenCloud	51957	5144 (9.9%)	4953 (9.5%)	3077 (5.9%)
M45	42825	6621 (15.5%)	6538 (15.3%)	4130 (9.6%)
Web Mining	822	369 (44.9%)	368 (44.8%)	255 (31.0%)

Table 3.2: Overall Statistics for Straggler Analysis: total number of jobs in the log, number of jobs longer than 5 min, and number of jobs with more than one map or reduce task.

- *Job configuration files*: Hadoop archives a job configuration file for each submitted job, which is an XML file that carries the specification of the MapReduce job (*e.g.*, the class names of the mapper and reducer, the types of the keys and values, the number of mappers, the number of reducers).
- *Job history files*: Hadoop also archives a job history file for each submitted job, which includes for each task the initialization time, start time, completion time, and time of each phase transition. In addition, this file includes a variety of counters, including the number of bytes/records read/written for each task.

Scope of Analysis: We limit our study to successfully completed jobs only, because counters and timing information were not captured in the job history for failed and killed jobs. Unless explicitly stated, we only consider jobs that run longer than five minutes because the impact of stragglers is insignificant for short jobs (*i.e.*, even jobs with stragglers are fast). A task only straggles with respect to some other task, so we only consider map and reduce phases that have at least two tasks. Table 3.2 summarizes the number of jobs and job phases analyzed in this section. When calculating the task runtime, we only take the map phase and reduce phase into account and ignore the *shuffle* and *sort* phases[5].

[5]Unfortunately, we cannot exclude the local sort phase of map tasks because the version of Hadoop running in all three clusters does not separate the timing of the local sort phase from that of the map phase.

Stragglers	Map	Reduce	Map ∧ Reduce	Map ∨ Reduce
OpenCloud	2967 (58%)	1940 (38%)	1109 (22%)	3798 (74%)
M45	3643 (55%)	2659 (40%)	1310 (20%)	4992 (75%)
Web Mining	140 (38%)	35 (9%)	18 (5%)	157 (43%)

Table 3.3: The number and fraction of jobs that have stragglers in map, in reduce, and both phases. More than 40% of jobs running longer than five minutes have at least one straggler.

3.3.2 Straggler Tasks

Dean *et al.* identified the straggler problem [37] and Ananthanarayanan *et al.* analyzed the problem in a production cluster [7]. In this subsection, we replicate the straggler analysis by Ananthanarayanan *et al.* [7][6]. Ananthanarayanan *et al.* categorized a task as a straggler if the task is running at least 50% longer than the median task in the same phase of the same job. Using this definition, the three research clusters are similar to the enterprise clusters studied by Ananthanarayanan [7] and an earlier study of the M45 cluster [75]: there are many stragglers, and some are orders of magnitude slower than the median runtime.

Table 3.3 summarizes the number of jobs that have stragglers in the map phase, the reduce phase, and in both phases. Overall, more than 40% (and up to 75%) of jobs that run longer than five minutes have at least one straggler. We also find that even the map phase, which is usually regularized by assigning a fixed amount of bytes to each task, frequently experiences the straggler problem. This finding confirms that the input data size alone is not a good indicator of task runtime in these three clusters. The same finding was reported in the analysis of enterprise clusters [29].

Knowing that many jobs have stragglers, we can also ask how many tasks per job are stragglers. To answer this question, we plot the cumulative distribution function of stragglers in each cluster and for each MapReduce phase. Figure 3.6 shows the results. In

[6]We use the term *straggler* instead of *outlier* for consistency with the original MapReduce paper [37].

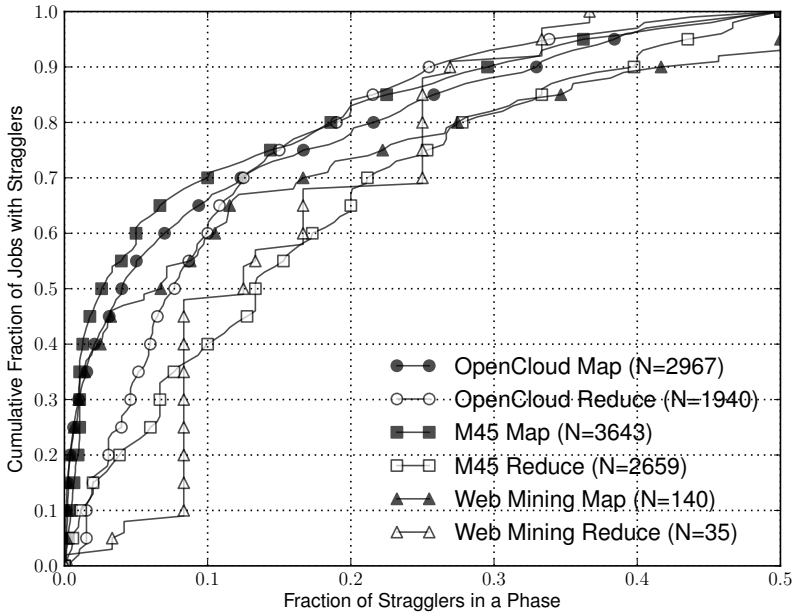
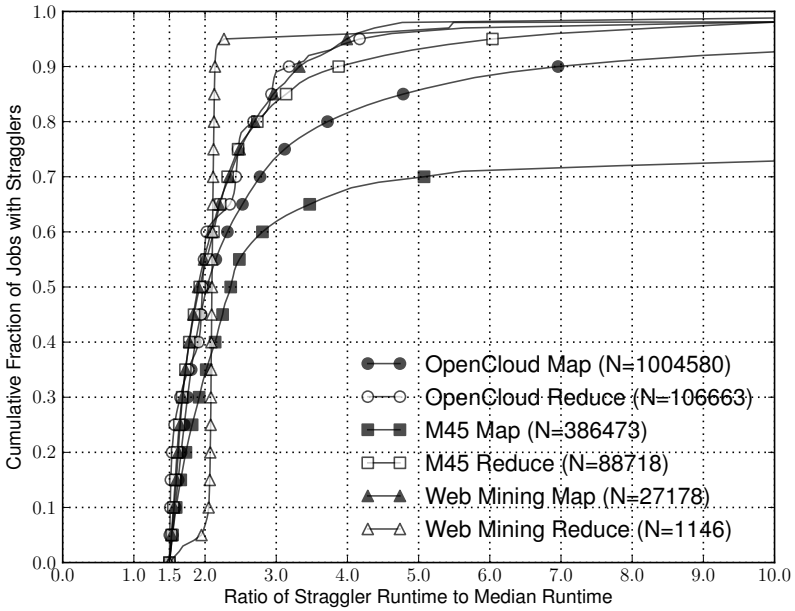


Figure 3.6: Cumulative distribution of the fraction of straggler tasks in Map and Reduce phases. N is the number of jobs that run for longer than five minutes and have stragglers per cluster, per phase.

all three clusters, 30% of map and reduce phases that include any stragglers have more than 10% of tasks as stragglers. The M45 reduce distribution closely resembles the one reported in Ananthanarayanan *et al.* [7]. However, we can observe variations in the distributions across phases and clusters.

In Figure 3.7, we show the distribution of relative runtimes of straggler tasks with respect to the median task runtime of the same phase in the job. In all three clusters, 75% of reduce-side stragglers complete within 2.5x of the median runtime. Map stragglers have larger variations across clusters: 55% and 65% of map straggler tasks complete within 2.5x of median runtime in M45 and OpenCloud respectively. As observed in Ananthanarayanan *et al.* [7], the distribution is heavy-tailed. We also show the values at 99%, 99.9%, and 100% percentiles in the table. Some straggler tasks run orders of magnitudes longer than the median runtime.



Percentile		99	99.9	100
OpenCloud	Map	70.0	1106.0	23068.5
	Reduce	15.3	64.9	54692.5
M45	Map	15.3	153.0	1537.5
	Reduce	11.4	143.8	1267.4
Web Mining	Map	11.2	66.9	170.7
	Reduce	46.4	404.3	409.3

Figure 3.7: Cumulative fraction of ratio of straggler runtime to median task runtime. N is the number of straggler tasks per cluster, per phase. The table shows straggler runtime at specific percentiles.

In summary, the straggler problem is prevalent in all clusters, and slow tasks frequently run more than 2.5x to orders of magnitude slower than the median.

3.3.3 Input Data Size and Task Runtime

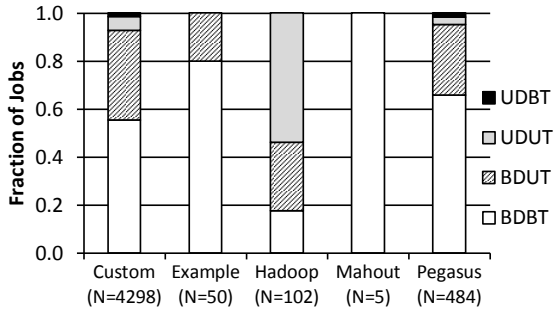
There are many causes of stragglers but one of the causes is *data skew*: some tasks take longer simply because they have been allocated more data. Ideally, if we assign the same amount of input data to all tasks, then all tasks should take the same time to execute. We evaluate how close the workloads are to this ideal case by analyzing the relationship between input data size and task runtime.

Method: For each phase of each job, we compute the ratio of the maximum task runtime in the phase to the average task runtime in that phase. We classify phases where this ratio is greater than 0.5 (meaning that at least one straggler took twice as long to process its data as the average) as *unbalanced in time (UT)*. Otherwise, the phase is said to be *balanced in time (BT)*. We compute the same ratio for the input data and classify phases as either balanced or unbalanced in their data (BD or UD).

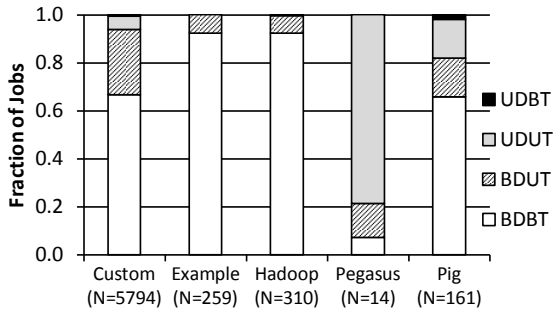
Map Input Record: Figure 3.8 shows the result for the map phases. We group the jobs by application types (Table 3.4) then breaks the results into four different types based on whether the input data and/or the runtime are balanced or not (*i.e.*, (U)BD(U)BT as defined in the previous paragraph).

As expected for the map phase, most jobs are balanced with respect to data allocated to tasks. However, a significant fraction of jobs, more than 20% for all but Mahout in the OpenCloud cluster, remain unbalanced in runtime and are categorized as BDUT. These results agree with the result of the previous study in enterprise clusters [29]. Mahout is effective at reducing skew in the map phase, clearly demonstrating the advantage of specialized implementations. Interestingly, Pig produces unbalanced map phases similar to users' custom implementations. Overall, allocating data to compute nodes simply based on data size alone is insufficient to eliminate skew.

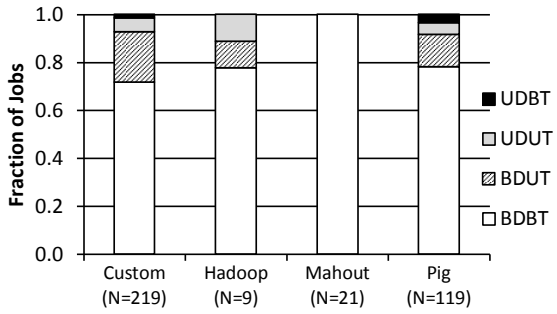
In Hadoop, the InputFormat mechanism is responsible for generating InputSplits that describe the input data processed by a map task. In our traces, less than 1% of jobs attempt to optimize the partitioning of the map input by using a custom input format. In addition, the user can manually specify the number of bytes per split if the input is stored in HDFS.



(a) OpenCloud



(b) M45



(c) Web Mining

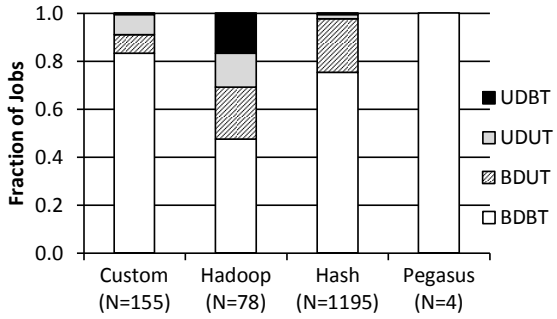
Figure 3.8: Distribution of map runtime with respect to # of input records per application. N is the number of successful jobs that run map functions in each category. The labels indicate the category: (U)BD(U)BT stands for (un)balanced input data, (un)balanced runtime.

Type	Description
Hadoop	Functions that are part of Hadoop MapReduce framework (*e.g.*, Identity, Regex).
Example	Functions from the example applications distributed with the Hadoop software (*e.g.*, Terasort, Wordcount, and LoadGen).
Mahout	Functions in Mahout jobs. Mahout is a machine learning algorithm package built on top of Hadoop [127].
Pegasus	Functions used in Pegasus jobs. Pegasus provides high-level APIs for large scale graph analysis [74]. The functions are either provided by the Pegasus framework or written by users.
Pig	Functions used in Pig jobs. Pig translates a Pig-Latin script into a DAG of MapReduce jobs [52, 104]. The functions include relational operators provided by the Pig execution engine as well as user-defined operators.
Custom	The remaining functions in the workload

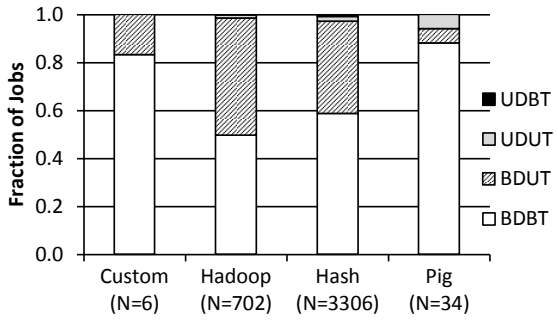
Table 3.4: Types of Map Functions in Figure 3.8

In our traces, 12% of all jobs from ten users in OpenCloud used this optimization. In M45 and Web Mining, less than 1% of all jobs from one and three users used the optimization, respectively. It is clear that users only rarely exploit these opportunities for optimizing the data allocation.

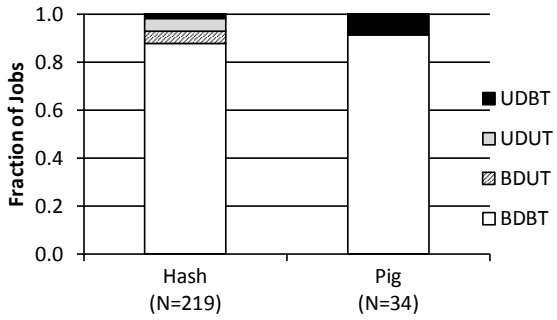
Reduce Key: We perform a similar analysis for the reduce phase by using the number of reduce keys as the input measure. Instead of application types, we group the jobs by the partition function employed (Table 3.5). The partition function is responsible for redistributing the reduce keys among the reduce tasks. Again, we find that users rely primarily on the default hash-partition function (Figure 3.10) rather than trying to optimize



(a) OpenCloud



(b) M45



(c) Web Mining

Figure 3.9: Distribution of reduce runtime with respect to # of reduce keys per partition function. The labels indicate the category: (U)BD(U)BT stands for (un)balanced input data, (un)balanced runtime.

Type	Description
Hash	The default hash partition function in Hadoop
Hadoop	Partition functions except hash provided by the Hadoop MapReduce framework (*e.g.*, TotalOrderPartitioner, KeyFieldBasedPartitioner)
Pegasus	Partition functions provided by the Pegasus framework [74]
Pig	Partition functions provided by the Pig execution engine [52, 104]
Custom	The remaining partition functions in the workload

Table 3.5: Types of Partition Functions in Figure 3.9

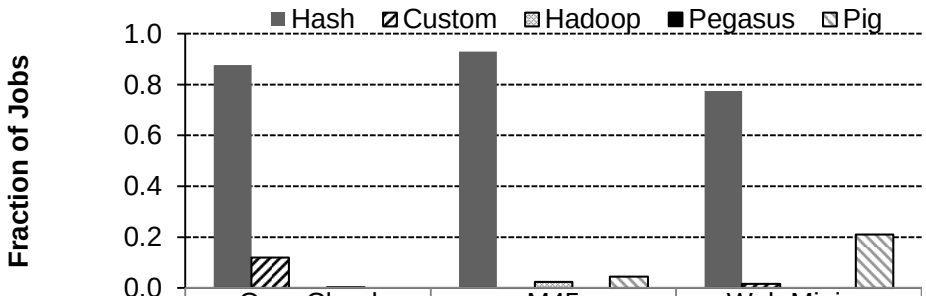
the data allocation manually to reducers.

Figure 3.9 shows the analysis per partition function[7]. Overall, hash partitioning effectively redistributes reduce keys among reduce tasks for more than 92% of jobs in all clusters (*i.e.*, BDBT+BDUT). Interestingly, we observed that as many as 5% of all jobs in all three clusters experienced the *empty reduce* problem, where a job has reduce tasks that processed *no* data at all due to either a suboptimal partition function or because there were more reduce tasks than reduce keys. (We observed the latter condition in 1% to 3% of all jobs.)

For the jobs with a balanced data distribution, the runtime is still unbalanced (*i.e.*, BDUT jobs) for 22% and 38% of jobs in the OpenCloud and M45 clusters, respectively. In both the OpenCloud and M45 clusters, custom data partitioning is more effective than the default scheme in terms of balancing both the keys and the computation. Other partitioning schemes that come with Hadoop distribution do not outperform hash partitioning in terms of balancing data and runtime. A noticeable portion of UDBT jobs (in which data are not balanced but runtime is balanced) use the total order partitioner, which tries to balance the keys in terms of the number of values. Pig, which uses multiple partitioners, consistently performs well in both M45 and Web Mining clusters.

In summary, we recommend pursuing techniques that automatically reduce skew to

[7]In the M45 workload, the number of reduce keys was not recorded for the jobs that use the new MapReduce API due to a bug in the Hadoop API. The affected jobs are not included in the figure.



	OpenCloud (N=50307)	M45 (N=36616)	Web Mining (N=749)
Hash	0.877	0.930	0.774
Custom	0.119	0.001	0.016
Hadoop	0.003	0.024	0.000
Pegasus	0.001	0.000	0.000
Pig	0.000	0.045	0.210

Figure 3.10: Fraction of jobs per partition function. The hash partitioning is dominant in all three clusters. The fraction of jobs using customized partition function is relatively small, less than 10% in all clusters.

achieve better overall performance given: a) the prevalence of load imbalance problems, b) the evidence that achieving load balance requires more than uniform data distribution, and c) users' reluctance to use the manual tuning features provided by Hadoop [84, 135].

3.3.4 Speculative Execution

Finally, we analyze the effectiveness of speculative execution. Speculative execution is the default mechanism implemented in Hadoop to handle straggler tasks. The intuition is that the task scheduler will eagerly schedule a redundant copy of a task whenever there exists an idle slot in an attempt to improve job completion time. The detailed implementation varies from version to version in Hadoop.

Method: For all successful jobs, we extract tasks that involve multiple attempts and categorize the tasks according to the characteristics of the attempts. The *Recompute* label indicates that the attempts were necessary to recover from the failure of the original attempt. The *Successful* label shows that the attempts were part of a successful speculative execution

		Jobs	Tasks	Default
OpenCloud	Map	16857 (32%)	548993 (3%)	Enabled
	Reduce	6751 (13%)	124931 (5%)	
M45	Map	1195 (3%)	6025 (0.1%)	Disabled
	Reduce	2165 (5%)	38835 (2%)	
Web Mining	Map	467 (57%)	6966 (1%)	Enabled
	Reduce	278 (34%)	630 (5%)	

Table 3.6: Total number of jobs and speculated tasks and default setting for speculative execution.

(*i.e.*, the speculative task was initiated after the original attempt but completed first). The *Unsuccessful* label represents an unsuccessful speculation (*i.e.*, the original attempt completed faster than speculative attempts). Since Hadoop does not keep track of precise timings for failed and killed attempts, we infer the missing data using other information. In particular, the failure of an original task attempt may have left the start time of the failed attempt unrecorded in the attempt log. In this case, we recover the start time from the task log.

Overall Statistics: In Table 3.6, we show whether the speculative execution is enabled by default for each cluster (if it is disabled by default, a user has to manually enable it for each MapReduce job), the number of all jobs, and the number of speculated tasks that triggered speculative execution. The OpenCloud and Web Mining clusters enable the speculative execution by default while M45 does not. Thus, all speculative executions observed in the M45 cluster were initiated by users who explicitly overrode the default value. The fraction of jobs and tasks that have at least one task attempted multiple times varies from cluster to cluster and between map and reduce. The fraction of jobs in M45 is significantly less than that of OpenCloud and Web Mining due to the default setting.

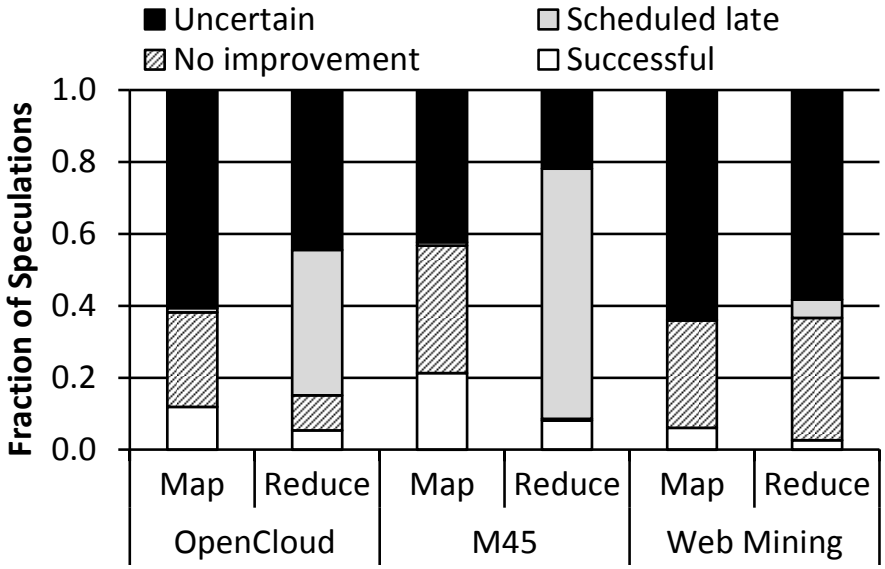


Figure 3.11: Fraction of Speculation Result per Phase per Cluster. In all clusters, a speculative execution completed faster than the original attempt (*successful*) at most 21% of the time. In all clusters, a large fraction of map speculations did not run significantly faster than the original attempt (*no improvement*). In OpenCloud and M45, many reduce speculations were *scheduled late* (speculative attempts ran only less than 10% of original attempt time before killed).

Figure 3.11 summarizes the results of speculations in all clusters. Speculative execution was not very successful in all three clusters: only 3 to 21% of speculations were *successful*. We further analyze those *unsuccessful* speculations by classifying each unsuccessful attempt as *scheduled late*, *no improvement*, and *uncertain*. A speculation is *scheduled late* if all speculation attempts for a task ran less than 10% of the original attempt runtime before being killed. Those killed attempts ran too briefly and so did have enough time to run to the end. Also, a speculation may not improve runtime at all. We label such speculations as *no improvement* when the oldest speculation attempt ran more than 90% of the original attempt runtime. The tasks in this category are likely to be genuinely long-running tasks, so re-executing the same tasks on different nodes does not improve their runtime significantly. We label the remaining unsuccessful speculations as *uncertain*.

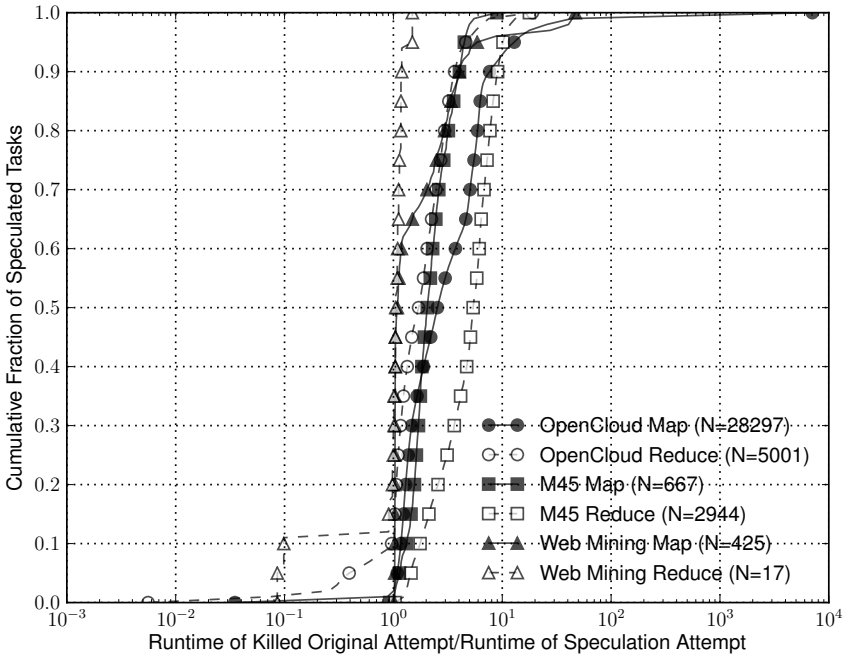


Figure 3.12: Successful speculative execution. Cumulative distribution of runtime ratio between the original attempt and successful speculation attempt. Median speedup of successful speculative execution varies from 1.05x to 5.4x and maximum speed up varies from 1.5x to 7000x.

In all three clusters, most of map speculations do not run significantly faster than the original attempts. For reduce, many speculation attempts in both OpenCloud and M45 are scheduled too late.

By analyzing *recomputed* tasks, we find only one instance of lost map output (*i.e.*, re-execute previously completed map task of which output is lost due to a node failure) in OpenCloud. Ananthanarayanan *et al.* also reported that such recomputation due to loss of map output is rare [7].

Successful Speculative Execution: In Figure 3.12, we analyze the speedup of successful speculative executions. We adjust the start time of the original reduce attempt to the time when the last map attempt completed if the reduce attempt started before com-

pletion of all map tasks because the shuffle time of such attempts includes a long waiting time for the completion of the map phase. In contrast, the speculative tasks immediately begin to shuffle because all input data are ready. Due to this adjustment, the ratio of some reduce attempts may be underestimated. Median speedup of speculative execution varies from 1.05x to 5.4x and 90% of the speculative execution runs up to 10x faster than the initial attempt. A few speculations run orders of magnitude faster than their initial attempts, and most of them complete within a minute. After closely investigating speculative executions that run orders of magnitude faster than their initial attempts, we suspect that those tasks are typically experiencing I/O problems (*e.g.*, very little disk and network I/O for an extended period time compared with the successful attempt) and the speedup was more pronounced especially when the job had a large number of tasks. Speculation was thus delayed at the end of each phase.

To sum up, the straggler tasks are prevalent and delay the job completion significantly in the three real world clusters. The speculative execution, the de facto strategy to handle stragglers, is rarely effective: only less than 21% of speculations were effective in the three clusters. When it is successful, however, the speed up is dramatic (up to 7000x). To improve the effectiveness of speculation, it is promising to make more informed decisions based on more accurate progress estimators and monitoring not only the tasks but also cluster resources [7, 98]. Also, more efficient skew mitigation strategies such as stealing work from stragglers rather than starting from scratch are important [84, 135].

3.4 Best Practices

We present a survey of best practices to mitigate skew in a MapReduce job derived from our observations in Section 3.2 and Section 3.3. We present them in order of our estimate of their implementation complexity. We discuss their benefits, their limitations.

Best Practice 1 *Use domain knowledge when choosing the map output partitioning scheme if the reduce operation is expensive: Range partition or some other form of explicit partition may be better than the default hash-partition.*

The default hash-partitioning scheme on key is a well-known technique in the parallel database literature for ensuring an even data distribution [40]. However, when the reduce operation is expensive and susceptible to skew, this simple technique often fails. Frequently, load must be balanced not at the granularity of keys but at the granularity of values as shown in Figure 3.5. In case of *Holistic Reduce* operations, the partitioning strategy must be application-dependent, which puts a significant burden on the developer.

To choose or implement better domain-specific partitioning strategies, the user must already be familiar with the properties of the application and the data. We found that the next best practice is useful in achieving this goal.

Best Practice 2 *Try different partitioning schemes on sample workloads or collect the data distribution at the reduce input if a MapReduce job is expected to run several times.*

Best Practice 3 *Implement a combiner to reduce the amount of data going into the reduce-phase and, as such, significantly dampen the effects of any type of reduce-skew.*

In MapReduce (and in Hadoop), at the output of the map phase and before the reduce phase, one can optionally implement a *combiner* that pre-aggregates the map output. Combiners are in general beneficial when the expected reduction ratio for data to be shuffled is significant.

The combiner optimization, however, may hurt performance if the CPU and disk I/O cost of the combiner is greater than the diminished network I/O cost [62]. As shown by Lin and Schatz [90], manually combining the output within a map is desirable, if possible, because it avoids extra serialization overheads to prepare the input for the combiner. This is true for the current Hadoop implementation. Other MapReduce and future Hadoop implementations may not have this issue.

Combiners are effective at handling *Partitioning Skew* and *Expensive Input* at the reduce side when the skew observed during the reduce phase is mainly due to the volume of data transferred during the shuffle phase because a proper combiner can significantly reduce the transferred data size and mitigate the problems. To be more effective, we recommend using both a combiner and domain-specific partitioning strategy as described in the previous practice.

Best Practice 4 *Use a pre-processing MapReduce job that extracts properties of the input data in the case of a long-running, skew-prone map phase. Appropriately partitioning the data based on the extracted properties before the real application runs can significantly reduce skew problems in the map phase.*

For MapReduce jobs that may experience *Expensive Input* on the map side and possibly a *Non-homomorphic Map*, skew can be eliminated by changing the allocation of input data to map tasks. If the behavior of the algorithm is known, then the user can run a separate MapReduce job that checks whether map-skew will occur. For example, a PageRank MapReduce preprocessing job can check whether there are graph nodes with large numbers of edges, and adjust the data partition accordingly, before executing the real PageRank job.

Best Practice 5 *Design algorithms whose runtime depends only on the amount of input data and not the data distribution.*

For MapReduce jobs with either *Holistic Reduce* or *Non-homomorphic Map* problems, the best approach to avoid skew is to re-design the map or reduce algorithms such that their runtime performance depends only on the size of the input data rather than the data value distribution. However, such redesign often requires extra expertise. For example, in the friends-of-friends clustering algorithm, we successfully eliminated skew by reimplementing the in-memory spatial index structure in Chapter 2. With the re-design, skew can be handled straightforwardly by assigning the same amount of data per node. However, it is not certain that such re-design is possible for all algorithms.

3.5 Conclusion

MapReduce and its open source implementation Hadoop have made large scale data analysis widely accessible. Such runtime systems free users from problems associated with distributed coordination, fault-tolerance, and scalability. However, users may still suffer from performance problems related to skew if they are not careful regarding their map and reduce implementations and how data is partitioned across tasks. In this chapter, we surveyed five common sources of skew in real MapReduce applications and demonstrated skew

problems as well as the effectiveness of current practices (*i.e.*, data partitioning, speculative execution) using real workloads from three Hadoop clusters. We presented five best practices in developing a MapReduce application to address these problems. In the following two chapters, we propose two approaches that systematically address the problems.

Chapter 4

SKEWREDUCE: COST-BASED PARTITION OPTIMIZATION

As we showed in Chapters 2 and 3, skew frequently occurs in real workloads. In our study, when a task experienced skew, the runtime of that task was sometimes orders of magnitude larger than the average runtime of peer tasks. In this chapter, we propose SkewReduce, a cost-based partition optimization technique that mitigates skew for a particular class of applications for which traditional approaches fail.

The standard approach to handling skew in parallel systems is to assign an equal number of data values to each partition via hash partitioning or clever range partitioning (see Chapter 6). These strategies effectively handle *partition skew*, which occurs when some nodes are assigned more data than others. For example, in MapReduce, the default partitioning strategy for the map phase is to assign one fixed-size block of the input data file to each map task so that all map tasks process the same amount of data. For the reduce phase, the default strategy is to hash-partition on the key attribute. In Section 3.3.3, we showed that both strategies effectively balanced the amount of input data per task, but a significant fraction of tasks still experienced skew. These tasks thus took significantly longer to process their data than other tasks even though they were assigned the same amount of data. In some cases, the cause is a hardware malfunction or high-load condition, and here existing techniques such as speculative execution work well. In many cases, however, we found speculative execution to be ineffective, indicating that the UDO computation time was simply non-uniform (*i.e.*, cost per input byte or per invocation was not a constant), leading to skew. (See the portion of '*UT' labels in Figure 3.8 and 3.9).

Given a UDO with non-uniform computation, finding the right granularity of data partitioning is challenge. If the granularity is too coarse (*i.e.*, a small number of large partitions), there may not be enough tasks to keep the cluster busy when a task experiences skew. If the granularity is too fine, we can still keep the cluster busy with unscheduled tasks at the cost

of extra overhead (*e.g.*, scheduling and/or reconciliation overheads due to extra tasks, as we discuss further in Section 4.2.1). The right granularity may vary between datasets and clusters, so the users typically rely on the system's decision (*e.g.*, the number of partitions is a multiple of either the number of nodes in the cluster or the number of blocks holding the input data file), which does not take into account the variability between datasets. More experienced users may guess the right granularity based on rules of thumb or past executions, which still burden users to choose the right parameter.

The key insight behind SkewReduce is that users know the details of their computation, and, while they are not parallel data processing experts, they can reason about the complexity of their algorithms. Based on this observation, SkewReduce asks the user for *cost models*, which are additional black-box functions that users implement to specify the complexity of their computations. Given user-defined cost models, a sample of the input dataset, the task scheduling algorithm, and the number of nodes in the cluster, SkewReduce generates an optimized data partitioning plan (*i.e.*, an allocation of the input data to different tasks for processing) as well as a scheduling order expected to yield a good runtime. The user-defined cost model captures the non-uniform computation of the UDO; the model estimates the cost of processing a given sample input data partition. The optimizer greedily searches the partition plan in a top-down manner by evaluating whether a split of the most expensive partition improves the estimated runtime. Since the optimizer can run offline, users can test different cluster configurations on a workstation before running the analysis in a cloud service such as Amazon EC2 [5] or Microsoft Windows Azure [140].

The current SkewReduce API and optimizer are designed for feature extracting applications (see Section 4.1 for examples). We chose this specific class of applications because it represents some important scientific data analyses and requires non-trivial data partitioning over a multi-dimensional space to reduce the impact of skew. However, the technique can be generalized to other applications and other parallel execution engines in a straight forward manner as we discuss in Section 7.1.

The SkewReduce system executes a UDO written following the SkewReduce API by transforming the computation into a set of Hadoop MapReduce jobs. SkewReduce is thus a new system for the parallel and skew-resilient execution of feature-extracting applications.

SkewReduce system is implemented on top of Hadoop. In experiments on real applications and real data, we show that SkewReduce improves the job completion time by a factor of up to 8 compared with a non-optimized implementation (*i.e.*, a default MapReduce partitioning strategy) without any code-level optimizations nor cluster configuration tunings. Also, we show that more accurate cost models yield better partition plans, but the cost model need not perfectly capture every aspect of the UDOs.

In this chapter, we first introduce feature extracting applications with examples (Section 4.1). We then present the SkewReduce API (Section 4.2.1) and the static optimization with user-defined cost models (Section 4.2.2 and 4.2.3). We demonstrate the efficacy of the optimization on real data from two different science domains (Section 4.3).

4.1 Feature Extracting Applications

We begin by describing three motivating applications from different scientific domains. The applications have a common structural pattern that we call *feature extraction*. We discuss the challenges that arise when trying to implement them on a MapReduce-type platform.

- **Astronomy Simulation**. Cosmological simulations are used to study the structural evolution of the universe on distance scales ranging from a few million light-years to several billion light-years. In these simulations, the universe is modeled as a set of particles. These particles represent gas, dark matter, and stars and interact with each other through gravity and fluid dynamics. Every few simulation timesteps, the simulator outputs a *snapshot* of the universe as a list of particles, each tagged with its identifier, location, velocity, and other properties. The data output by a simulation can thus be stored in a relation with the following schema:

  $$\texttt{Particles}(id,\ time,\ x,\ y,\ z,\ v_x,\ v_y,\ v_z,\ \cdots)$$

  State of the art simulations (*e.g.*, Springel *et al.* [122]) use over 10 billion particles producing a data set size of over 200 GB per snapshot and are expected to significantly grow in size in the future.

  Astronomers commonly used various sophisticated clustering algorithms [53, 79, 119] to recognize the formation of interesting structures such as galaxies. The clustering

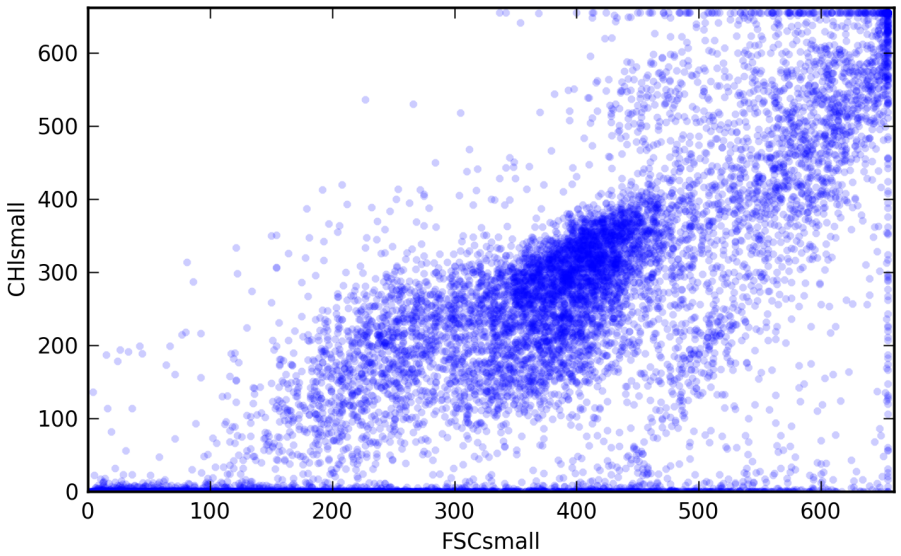


Figure 4.1: A scatter plot of flow cytometry measurements. Each point represents an organism and clusters represent populations. The axes correspond to different wavelengths of light.

algorithm is typically executed on one snapshot at a time [85]. Given the size of individual snapshots, however, astronomers would like to run their clustering algorithms on a parallel data processing platform in a shared-nothing cluster.

- **Flow Cytometry**. A flow cytometer measures scattered and fluoresced light from a stream of particles, using data analysis to recognize specific microorganisms. Originally devised for medical applications, it has been adapted for use in environmental microbiology to determine the concentrations of microbial populations. Similar microorganisms exhibit similar intensities of scattered light, as in Figure 4.1.

  In an ongoing project in the Armbrust Lab at the University of Washington [11], flow cytometers are being continuously deployed on ocean-going vessels to understand the ocean health. All data is reported to a central database for ad hoc analysis and takes the form of points in a 6-dimensional space, where each point represents a particle or

organism in the water and the dimensions are the measured properties.

As in the astrophysics application, scientists need to cluster the resulting 6D data. As their instruments increase in sophistication, so does the data volume, calling for efficient analysis techniques that can run in a shared-nothing cluster.

- **Image Processing**. As a final example, consider the problem of analyzing collections of 2D images. In many scientific disciplines, scientists process such images to extract objects (or features) of interest: galaxies from telescope images, hurricanes from satellite pictures, etc. As these images grow in size and number, parallel processing becomes necessary.

General Feature Extracting Applications. Each of these scientific applications follow a similar pattern: data items (events, particles, pixels) are embedded in a metric space, and the task is to identify and extract emergent features from the low-level data (populations, galaxies). These algorithms then typically return (a) a set of features (significantly smaller than the input data), (b) a modified input dataset with each element tagged with the corresponding feature (potentially as large as the input), or (c) both. For example, the output of the astronomy clustering task is a list of clusters with the total number of particles in each and a list of the original particles annotated with their cluster identifier.

Parallel Implementation Challenges. A straightforward way to parallelize such feature extraction applications in a compute-cluster with N nodes is the following: (1) split the input into N equal-sized hypercubes, (2) extract features in each partition and annotate the input with these initial features, (3) reconcile features that span partition boundary, relabeling the input as appropriate. With existing parallel processing systems, there are several challenges with expressing this seemingly simple algorithm in a manner that achieves high performance.

First, the data distribution in many scientific applications is highly skewed. Even worse, the processing time of many feature-extraction algorithms depends not only on the number of data points but also on their distribution in space. For example, in a simple clustering algorithm used in astrophysics called "friends-of-friends" [33], clusters correspond to connected components of the graph induced by the "friend" relationship — two particles are

friends if they are within a given distance threshold. To identify a cluster, the algorithm starts with a single point, then searches a spatial index to find its immediate friends. For each such friend, the algorithm repeats the search recursively. In a sparse region with N particles, the algorithm completes in $O(N \log N)$ time (*i.e.*, all particles are far apart). In a dense region, however, a single particle can be a friend of all the other particles and vice versa. Thus, the algorithm takes $O(N^2)$ time. In the two simulation snapshots that we received from astronomers [85], we found that the number of friends associated with a given particle varied between 2 and $387,136$. As a result, without additional optimizations, a dense region takes much longer to process than a sparse one even when both contain the same number of total particles [85]. The consequence is a type of computational skew, where some data partitions require dramatically more time than others to process. Computational skew is the reason that the naïve parallel implementation of the astronomy clustering application in Chapter 2 required over 20 hours, while an optimized one took only 70 minutes on the same dataset [85]. Our key motivation is that *existing platforms do nothing to reduce computational skew.* In our case, developing a skew-resistant algorithm (by optimizing index traversal to avoid quadratic behavior in the dense region) required significant effort from multiple experts over several weeks [85].

Second, the feature reconciliation phase (which we refer to as the "merge" phase) can be both CPU and memory intensive. For example, to reconcile clusters at the boundary of two data partitions requires processing all particles within a small distance of that boundary. If the space is initially carved into N partitions, it may not be efficient or even possible for a single node to reconcile the data across all these partition boundaries in one step. Instead, reconciliation should be performed in a hierarchical fashion, reconciling increasingly large regions of the space, while keeping the amount of data to process at each step approximately constant (*i.e.*, the memory requirement cannot increase as we move up the hierarchy). At the same time, while the local data processing and later merge steps proceed, the input data must be labeled and re-labeled as necessary, *e.g.*, to track feature membership. While it is possible to implement both functions using existing systems, expressing them using current APIs is non-trivial.

T	A record in the original input data file assigned to a region (*e.g.*, a particle in an astronomy simulation)
S	A record set aside during the process phase or merge phase. (*e.g.*, a particle far away from a partition boundary tagged with a local cluster id).
F	An object representing a set of features extracted during the process phase for a given region. May not be relational. Includes enough information to allow reconciliation of features extracted in different partitions (*e.g.*, the clusters identified so far and the particles near a partition boundary)
Z	A record in the final result set (*e.g.*, a particle tagged with a global cluster id)

Table 4.1: Notations in Section 4.2

Problem Statement Summary. The goal of SkewReduce is to enable scientists to *easily express* and *efficiently execute* feature-extraction applications at very large scale without consideration of resource constraints and data or computation skew issues.

4.2 SkewReduce

SkewReduce has two components. The first component is an API for expressing spatial feature-extraction algorithms such as the ones above. We present the API in Section 4.2.1. The functions in our API are translated into a dataflow that can run in a MapReduce-type platform [37, 61, 68]. The second component of SkewReduce is a static optimizer that partitions the data to ensure skew-resistant processing if possible. The data partitioning is guided by a user-defined cost function that estimates processing times. We discuss the cost functions in Section 4.2.2 and the SkewReduce optimizer in Section 4.2.3.

4.2.1 Basic SkewReduce API

Informed by the success of MapReduce [37], the basic SkewReduce API is designed to be a minimal control interface allowing users to express feature extraction algorithms in terms of serial programs over familiar data structures. The *basic SkewReduce API* is the minimal

interface that must be implemented to use our framework. The basic API is

$$\begin{aligned} \mathsf{process} &: [T] \to (F, [S]) \\ \mathsf{merge} &: (F, F) \to (F, [S]) \\ \mathsf{finalize} &: (F, [S]) \to [Z] \end{aligned}$$

The notation used in these types is defined in Table 4.1. At a high-level, T refers to the input data. F is the set of features and S is an output data field that must be tagged with the features F to form Z. The above three functions lead to a very natural expression of feature extracting algorithms: First, partition the data (not shown). Second, apply process to each partition to get an initial set of local features and an initial field. Third, merge, or reconcile, the output of each local partition to identify a global set of features. Finally, adjust the output of the original process functions given the final, global structures output by merge. For example, in the case of the astronomy simulation clustering task, process identifies local clusters in a partition of the 3D space. merge hierarchically reconciles local clusters into global clusters. Finally, the finalize function relabels particles initially tagged by process with a local cluster ID using the appropriate global cluster ID.

The functions of the SkewReduce API loosely correspond to the API for distributed computation of algebraic user-defined aggregates found in OLAP systems and distributed dataflow frameworks. For example, Yu *et al.* propose a parallel aggregation framework consisting of functions initialreduce, combine, and finalreduce [146]. The function initialreduce generates intermediate partial aggregates, combine merges partial aggregates, and the final aggregate value can be further transformed by finalreduce.

The distinguishing characteristic of our API is that our analog of the initialreduce and finalreduce functions return two types of data: a representation of the extracted features, and a representation of the "tagged" field. A given algorithm may or may not use both of these data structures, but we have found that many do.

We now present the three functions in SkewReduce's API in more detail.

Process: Local Computation with Set-Aside

The process function locally processes a sequence of input tuples producing F, a representation of the extracted features, and $[S]$, a sequence of tuples that are set aside from the hierarchical reconciliation. In our astronomy simulation use-case, process performs the initial clustering of particles within each partition. Although we can forward all the clustering results to the merge function, only particles near the boundary of the fragment are necessary to merge clusters that span two partitions. Thus, process can optionally *set aside* those particles and results that are not required by the following merges. This optimization is not only helpful to reduce the memory pressure of merge but also improves overall performance by reducing the amount of data transferred over the network. In this application, our experiments showed that almost 99% of all particles can thus be set aside after the Process phase (Figure 4.9).

Merge: Hierarchical Merge with Set-Aside

The merge function is a binary operator that combines two intermediate results corresponding to two regions of space. It takes as input the features from each region and returns a new merged feature set. The two feature set arguments are assumed to fit together in the memory of one node. This constraint is a key defining characteristic of our target applications. This assumption is shared by most user-defined aggregate frameworks [105, 123, 146]. However, SkewReduce provides more flexibility than systems designed with trivial aggregation functions such as sum, count, average in mind. Specifically, we acknowledge that the union of all feature sets may not fit in memory, so we allow the merge function to set aside results at each step. In this way, we ensure that the size of any value of type F does not grow larger than memory. We acknowledge that some applications may not exhibit this property, but we have not encountered them in practice. We assume that both functions process and merge set aside data of the same form. This assumption may not hold in general, but so far we have found that applications either set aside data in the process phase or in the merge phase, but not both.

In our running example, the merge function combines features from adjacent regions of

space, returning a new feature object comprising the bounding box for the newly merged region of space, the cluster id mappings indicating which local clusters are being merged, and the particles near the boundary of the new region. Figure 4.2 illustrates the merge step for four partitions P1 through P4. The outer boxes, Pi, represent the cell boundaries. The inner boxes, I, are a fixed distance ϵ away from the corresponding edge of the region. The local clustering step, process, identified a total of six clusters labeled C1 through C6. Each cluster comprises points illustrated with a different shade of gray and shape. However, there are only three clusters in this dataset. These clusters are identified during the hierarchical merge step. Clusters C3, C4, C5, and C6 are merged because the points near the cell boundaries are within distance ϵ of each other. In Figure 4.2, C2 does not merge with any other cluster because all points in C2 are sufficiently far from P1's boundary. We can thus safely discard C2 before merging: These points are not needed during the merge phase. In general, we can discard all the points in the larger I regions before merging, reducing the size of the input to the merging algorithm. This reduction is necessary to enable nodes to process hierarchically larger regions of space without exhausting memory.

Finalize: Join Features with Set-Aside Data

The finalize function can be used to implement a join between the final collection of features and the input representation as output by the process and merge functions. This function is useful for tagging the original data elements with their assigned feature. The finalize function accepts the final feature set from the merge phase and a *single tuple* set aside during processing. The SkewReduce framework manages evaluation of this function over the entire distributed dataset.

Our emphasis on distinguishing "features" and "set aside" data may at first appear to be over-specialized to our particular examples, but we find the idiom to be quite general. To understand why, consider the analogous distinction between *vector* and *raster* representations of features. For example, Geographic Information Systems (GIS) may represent a geographic region as an image with each pixel assigned a value of "road", "waterway", "building", etc. (the raster representation). Alternatively, these objects may be represented

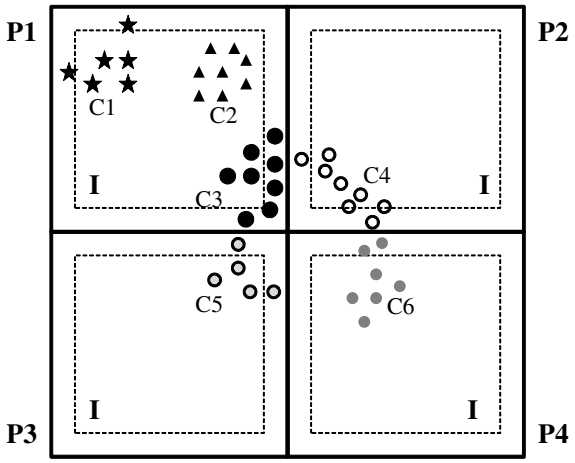


Figure 4.2: **Illustration of the merge step of the clustering algorithm in the SkewReduce framework.** Data is partitioned into four chunks. Points with the same shape are in the same global cluster. Point with different colors but with identical shapes are in different local clusters (*e.g.*, the circles in the middle of the figure). Each Pi labels the cell boundary and each I labels the interior region. Only points outside of I are needed in the subsequent merge phase. After the hierarchical merge phase, three cluster mappings are generated: (C4,C3), (C5,C3), and (C6,C3). Such mappings are used to relabel local cluster ids during the finalize phase.

individually by line segments, polygons, or some other complex object (the vector representation). Neither representation is ideal for all algorithms, so both are frequently computed and maintained. In our running example, the tagged particles are analogous to the raster representation — each point in the original dataset is labeled with the feature to which it contributes.

The user thus specifies the above three functions. Given these functions, SkewReduce automatically partitions the input data into hypercubes and schedules the execution of the process, merge, and finalize operators in a Hadoop cluster. We further discuss the details of the Hadoop translation in Section 4.3. The partition plan is derived by SkewReduce's optimizer, as we discuss below.

In many application domains the process function satisfies the following property:

Definition 4.2.1 process *Monotonicity* *For datasets R, S where $R \subseteq S$, the execution time* time[*process*(R)] $\leq$ time[*process*(S)] *(Intuition: as the dataset grows incrementally, so must the local processing cost).*

The SkewReduce's optimizer is designed primarily for applications where this property holds. However, it can still handle applications that violate this property, as we discuss in Section 4.2.3.

For the applications we encounter in practice, we find that process is far more expensive than merge, which causes aggressive partitioning to be generally beneficial. In these cases, the limiting factor in partitioning is the scheduling overhead. In contrast, if merge is expensive or comparable relative to process, partitioning simply ensures that no node is allocated more data than will fit in its memory.

Optional Pre-Processing. The process function operates on a set of records $[T]$. In some applications, especially those operating on arrays, individual records are not cells but rather small neighborhoods of cells, sometimes called *stencils*. This distinction is not an issue for process, which receives as input a contiguous block of cells and can thus extract stencil neighborhoods unilaterally. However, since the optimizer operates on a sample of the input data, SkewReduce must apply a pre-processing step that extracts application-defined *computational units* before sampling them. For this reason, although not part of the basic API, we allow a user to provide a custom function to transform a sequence of "raw" records into a sequence of computational units, $[T]$.

4.2.2 Cost Functions

We have presented the basic SkewReduce API, but we have not explained how skew is handled. Both the process and merge phases of the API are crucially dependent on the initial partitioning of data into regions. Feature extraction applications often exhibit both data skew and computational skew, and both are determined by how the data are partitioned. Datasets prone to significant data and computational skew (usually due to extreme variations in data density) can be processed efficiently if an appropriate partition-and-merge plan can be found. As we will show, plan quality can be improved dramatically if the user

can estimate the runtime costs of their process and merge functions.

We allow the user to express these costs by providing two additional cost functions C_p and C_m, corresponding to process and merge, respectively. These cost functions operate serially on samples of the original dataset returning a real number; that is:

$$C_p : (S, \alpha, B) \to \mathbb{R}$$
$$C_m : (S, \alpha, B) \to (S, \alpha, B) \to \mathbb{R}$$

where S is a sample of the input, α is the sampling rate, and B is a bounding hypercube.

The cost functions accept both a representation of the data (the sample S) and a representation of the region (the bounding hypercube B, represented as a sequence of ranges, one for each dimension). The cost of the feature extraction algorithms we target is frequently driven by the distribution of the points in the surrounding space. One approach to estimate cost inexpensively is therefore to build a histogram using the bounding hypercube and the sample data and compute an aggregate function on that histogram. The sampling rate α allows the cost function to properly scale up the estimate to the overall dataset. When discussing cost functions in the remainder of this chapter, we omit the bounding hypercube and sampling rate parameters when they are clear from the context.

Given a representative sample, the cost functions C_p and C_m must be representative of actual runtimes of the process and merge functions. More precisely, the functions must satisfy the following properties.

- **Fidelity.** For samples R, S, if $C_p(R) < C_p(S)$, then

  $$\text{time}[\mathsf{process}(R)] < \text{time}[\mathsf{process}(S)]$$

  (intuition: the true cost and the estimated cost impose the same total order on datasets). Similarly, for samples R, S, T, U, if $C_m(R, S) < C_m(T, U)$, then

  $$\text{time}[\mathsf{merge}(R, S)] < \text{time}[\mathsf{merge}(T, U)]$$

- **Boundedness.** For some constants ρ_p and ρ_m and samples R and S,

  $$\text{time}[\mathsf{process}(R)] = \rho_p C_p(R) \text{ and time}[\mathsf{merge}(R, S)] = \rho_m C_m(R, S)$$

For the boundedness condition, we can estimate the constant factors ρ_p and ρ_m in at least two ways. The first method is to run the process and merge algorithms over a data sample and compute the constants. This type of approach is related to curve fitting for UDF cost estimation [24]. The second method is to derive new constants for a new cost function from past executions of the same analysis.

Many MapReduce-style analytic systems are running on top of chunk-based distributed file systems such as GFS, HDFS, and S3 and use the chunk as a unit of task distribution and computation. SkewReduce takes a similar approach and requires that the process and merge functions have the ability to process at least one chunk-size of input data without running out of memory. Alternatively, we could optionally allow users to specify memory usage estimation functions that take a form analogous to the cost functions above. In both cases, the optimizer ensures a partition plan with sufficient granularity that no operator runs out of memory.

4.2.3 SkewReduce's Optimizer

There are two potential optimization goals for a SkewReduce application: minimize execution time or minimize resource usage. SkewReduce's current optimizer adopts a traditional approach and *minimizes the query execution time subject to a constraint on the number of available machines in a cluster*. This constraint can be dictated by the size of a locally available cluster or by monetary reasons when using a pay-as-you-go platform such as Amazon EC2 [5]. SkewReduce's optimizer could be used to try alternative cluster sizes if a user tries to find some desired price-performance trade-off, but we leave it for future work to automate such exploration.

SkewReduce's optimizer is designed to operate on a small sample of the entire dataset, so that the optimizer can execute on a user's desktop before the user acquires or even just reserves any resources on a large cluster. In this chapter, we do not address the problem of how the user generates such a sample. Such samples are already commonly used for debugging in these environments.

At a high level, SkewReduce's optimizer thus works as follows: given a sample S of

the input data, process and merge functions and their corresponding cost functions C_p and C_m, a compute cluster-size constraint of M nodes, and a scheduling algorithm, the optimizer attempts to find the *partitioning plan* that minimizes the total query execution time. The user-supplied cost functions and the scheduling algorithm guide the optimizer's search for such best plan. SkewReduce works best with a task scheduler that minimizes makespan subject to task dependencies. However, it uses the scheduler as a black box and can therefore work with various schedulers.

Since the scheduler is modeled as a black box and the cost functions may not be completely accurate, SkewReduce does not guarantee to generate an optimal plan. However, our experiments in Section 4.3 show that it finds very efficient plans in practice (Figure 4.3).

We begin by defining the SkewReduce partition plan, execution plan, and the optimization problem more precisely.

Partition Plan: A SkewReduce partition plan is a full binary tree where all intermediate nodes represent merge operators and all leaf nodes represent process operators. Each node in the tree is associated with a bounding hypercube defining the region of space containing all data in the partition. The hypercubes at a given height in the tree partition the space; there are no gaps or overlaps.

Valid Partition Plan: A partition plan is valid if no node in the plan is expected to receive more data than will fit in memory. The memory size is applied after scaling the sample data back to the original input data size, assuming the sample, S, is representative. For example, if a 1% data sample leads to a partition with $2,000$ particles, and we know that a single node cannot process more than $100,000$ particles, the plan will not be valid since $2,000 * 100 > 100,000$.

Execution Plan: A SkewReduce execution plan comprises a partition plan and its corresponding schedule using a job scheduling algorithm schedule. A valid execution plan is a valid partition plan and its schedule.

Optimization Problem: Given a sample S of the input data, process and merge functions with their corresponding cost functions and constants (ρ_p, ρ_m), a compute cluster of M nodes, a scheduling algorithm and constant operator scheduling delay (Δ), return the valid execution plan that is estimated to minimize query runtime.

Algorithm 3 Searching Optimal Partitioning Plan

Input: p_0: root partition
M: the number of machines
Output: P: the best partitioning plan
bestCost: the best cost to run P
schedule: the schedule of P

```
1:  P ← {p_0}
2:  bestCost ← schedule_cost(P, M)
3:  L ← {p_0}  // all leaf partitions in P
4:  while L ≠ ∅ do
5:      p ← choose the most expensive partition
6:      p_l, p_r ← find best split of p
7:      c ← ρ_p max{C_p(p_l), C_p(p_r)} + ρ_m C_m(p_l, p_r) + Δ
8:      force ← p does not satisfy memory requirement
9:      if c < ρ_p C_p(p) or force then
10:         s ← schedule_cost(P ∪ {p_l, p_r}, M)
11:         if s < bestCost or force then
12:             L ← L ∪ {p_l, p_r}
13:             P ← P ∪ {p_l, p_r}  // p becomes an internal node.
14:             bestCost ← s
15:         end if
16:     end if
17:     L ← L − {p}
18: end while
19: if all p ∈ P satisfies memory requirement then
20:     return (P, bestCost, schedule(P))
21: else
22:     print failed to find a valid plan
23: end if
```

Optimizing the Partition Plan

The search space of the optimizer is the set of all possible partitions of the hypercube defined by the input data. The optimizer enumerates potentially interesting partition plans in this search space using a greedy strategy. This greedy strategy is motivated by the fact that all process cost functions are assumed to be monotonic (Section 4.2.2).

Starting from a single partition that corresponds to the entire hypercube bounding the input data I, and thus also the data sample, S, the optimizer greedily splits the most

expensive leaf partition in the current partition plan. The optimizer stops splitting partitions when two conditions are met: (a) All partitions can be processed and merged without running out of memory; (b) No further leaf-node split improves the runtime: *i.e.*, further splitting a node increases the expected runtime compared to the current plan. Algorithm 3 summarizes this approach.

In order for a partition split to decrease the runtime, the savings in process processing times must outweigh the cost of the extra merge including scheduling overheads. More specifically, in the algorithm, the runtime after the split for the partition is estimated to be the sum of the runtime of the slower of the two new process operators (given that they will most likely be processed in parallel), the runtime of merge, and the task scheduling delay (Δ) for the merge operator. This is compared to the estimated runtime before the split, which was simply the time to run the process operator (line 9). Additionally, the resulting parallel execution plan must be valid and must improve the total estimated runtime. We estimate the total runtime by running the black-box scheduling algorithm. The algorithm updates the current best plan only when the runtime improves (line 10-11) or if a split is mandatory due to memory constraints. We perform this two-level filtering to reduce the number of calls to the scheduler function.

The optimizer returns an error if no valid partition plan exists. That is, if in all considered partition plans at least one process or one merge operators run out of memory (line 22).

The algorithm uses two key subroutines: finding the best point where to split a partition in two (line 5) and estimating the cost of a schedule in terms of processing time (line 10). In the following subsections, we discuss each of these two subroutines and SkewReduce's default implementation of these routines. Alternatively, the user may also supply custom implementations.

Partition Splitting

When splitting a hypercube in two, the optimizer has two choices to make: which axis to use for the split and at what point along this axis to perform the split.

Algorithm 4 Searching best split for a given partition

Input: B: bounding hypercube

S: sample data bounded by B

Output: *bestSplit*: axis and splitting point

1: $bestCost \leftarrow \infty$
2: $bestSplit \leftarrow null$
3: $A \leftarrow chooseAxes(B, S)$
4: **for all** $axis \in A$ **do**
5: $split \leftarrow$ find best split point along $axis$
6: $B_l, B_r \leftarrow$ split B at $split$ along $axis$
7: $c \leftarrow \max\{C_p(B_l, S), C_p(B_r, S)\}$
8: **if** $c < bestCost$ and satisfies merge memory requirement **then**
9: $bestSplit \leftarrow (axis, split)$
10: $bestCost \leftarrow c$
11: **end if**
12: **end for**
13: **return** *bestSplit*

An ideal split should partition the data into two subpartitions with identical real runtimes. In contrast, the worst split creates two subpartitions with very different real runtimes, with the runtime for the slower subpartition similar to the pre-split runtime.

Algorithm 4 shows the optimizer's approach to choosing the split axis and split point for a given partition. This algorithm applies the user-defined cost functions on the data sample, S, to estimate execution times.

For a low dimensional data, typically 3 to 4, the optimizer exhaustively tries to split the data along each of the available axes because the optimization process is low-overhead (as we show later in Figure 4.8). For a high dimensional data, the user can supply a heuristic to filter out bad split axes to improve optimization time. We define the best split to be the one that minimizes the maximum cost C_p of any of the subpartitions created without violating the merge memory requirement.

To select the point along an axis where to split the data, different algorithms are possible. We present and compare three strategies. All three methods require that the examined sample data be sorted along the splitting axis with tie-breaking using values in other dimensions. Thus, we sort the sample data before run the strategy.

Discrete: The Discrete approach considers splitting the data at each one of n uniformly-spaced points along the splitting-axis. n is given as a parameter. For each point, the discrete strategy computes the cost of splitting the data at that point. The discrete approach is thus the most general strategy because it can work even when the cost function is not monotonic. It simply tries all possible splitting points assuming a given minimum granularity. On the other hand, this strategy may not return the best estimated splitting point, especially if n is small.

Binary Search: This approach requires that cost functions be monotonic and performs a binary search for the best split point. The algorithm terminates after examining all $\log |S|$ candidate split points. Binary search always returns the optimal split as estimated by the cost function.

Incremental Update: The Incremental Update approach requires that the cost function be monotonic and incrementally updatable. That is, whenever the cost function is updated with a sample through an API call, the new cost is returned. Given these restrictions, the Incremental Update approach achieves the best optimization performance. The approach searches for the best split point in two phases. The algorithm starts with two empty subpartitions. It continuously adds samples to these subpartitions starting at both ends of partitioning axis. Each new data point is added to the partition currently estimated to have the lower runtime. The algorithm terminates when all samples have been assigned to a subpartition and the splitting point is the mid-point between the last sample inserted into each partition.

If multiple points fall on the partition boundary, the algorithm enters a second phase, where it computes the fraction of such points that were assigned to each partition. At runtime, when the entire dataset is partitioned, points on the same partition boundary are randomly distributed to subpartitions according to these precomputed proportions.

Estimating the Cost of a Schedule

The newly split partitions are only added if the candidate plan yields a better total runtime than the current plan. We estimate the runtime by calling a black box scheduling function schedule. To match the units of the operator costs to those of the scheduling overheads, we scale the process and merge costs using the pre-computed ρ_p, ρ_m constants, thus converting these costs into time units.

Converting a schedule to a cost estimate is straight forward; we invoke the scheduling algorithm with the costs of all operators and M slots as input then take the total runtime. While we leave the scheduling algorithm as a black box, we found that Longest Processing Time (LPT) scheduling algorithm [56] works well in practice and satisfies all necessary features such as job dependency and multiple slots. Thus, we use LPT algorithm in the prototype.

4.3 Evaluation

In this section, we evaluate the performance of SkewReduce on the friends-of-friends clustering task over datasets from two different domains: astronomy and oceanography (see Section 4.1). Table 4.2 summarizes the properties of the two datasets. We implemented friends-of-friends in a straightforward fashion without any optimizations, and using a standard KD-tree for storing local data and looking up friends.

Summary. We answer the following questions: (1) Does SkewReduce improve task completion times compared to uniform data partitioning, and, if so, is the difference significant? (2) How important is the fidelity of the cost model for SkewReduce's optimization? (3) How does the sample size affect cost estimates and ultimately performance? (4) What is the overhead of scheduling and optimization in SkewReduce? Our results show that SkewReduce imposes a negligible overhead (Figure 4.8) and can decrease total runtime by a factor of 2 or more compared to uniform data partitioning (Figure 4.3). We also find that small sample sizes of just 1% suffice to guide optimization, but the quality of the resulting plan does depend on the characteristics of the sample (Figures 4.6 and 4.7). Finally, a cost function that better captures the analysis algorithms helps SkewReduce find better

plans, but even an approximate cost function can improve runtime compared to not using SkewReduce at all (Figures 4.4 and 4.5).

Implementation. The SkewReduce prototype consists of two Java classes: the SkewReduce optimizer and the SkewReduce execution engine. The optimizer takes the cost model and sample data as input and produces an optimized partition plan and a corresponding schedule. The execution engine converts the plan into a graph of Hadoop jobs and submits them to Hadoop according to the schedule from the optimizer. SkewReduce deploys a full MapReduce job for the initial data partitioning task (if necessary) and for each finalize operator, but deploys a map-only job for each process or merge operator. This design gives us better control over the timing of the schedule because Hadoop only supports user specified priorities at the job level rather than at the task level.

SkewReduce minimizes the scheduling overhead by using asynchronous job completion notifications of the Hadoop client API. Optionally, the user can implement the finalize operator as a Pig script [104] instead of a MapReduce program.

Setup. We perform all experiments in an eight-node cluster running Hadoop 0.20.1 with a separate master node. Each node uses two 2 GHz quad-core CPUs, 16 GB of RAM, and two 750 GB SATA disk drives (RAID 0). All nodes are used as both compute and storage nodes. The HDFS block size is set to 128 MB and each node is configured to run at most four map tasks and four reduce tasks concurrently.

We compare SkewReduce to various uniform data partitioning algorithms. We use the LPT scheduling algorithm for the SkewReduce optimizer. Uniform alternatives cannot use this approach because they do not have any way to estimate how long different tasks will take to process the same amount of data.

Default Optimization Parameters. SkewReduce's optimizer assumes a MapReduce job scheduling overhead (Δ) of 10 seconds [108]. Unless indicated otherwise, experiments use a sample size of 1%. The default cost function builds a 3D equi-width histogram of the data. Each bucket covers a range equal to the friend distance threshold along each dimension. The cost is computed as the sum of squared frequencies for all buckets. Each

Dataset	Size	# of items	Description
Astro	18 GB	900 M	Cosmology simulation
Seaflow	1.9 GB	59 M	Flow Cytometry

Table 4.2: Datasets used in the evaluation

	Data Size			Histogram 1D			Histogram 3D		
Astro	83	4.3	10^{-6}	1500	2.9	10^{-12}	3.0	40	10^{-7}
Seaflow	4.8	1.6	10^{-5}	9.3	130	10^{-12}	6.0	200	10^{-8}

Table 4.3: Cost-to-time conversion constant for cost models (ρ_p, ρ_m,scale)

frequency is scaled back by the sample size (*e.g.*, for a 1% sample, all bucket frequencies are multiplied by 100) before squaring. The intuition behind this cost model is this: To identify a cluster, the friends-of-friends algorithm starts with a point and recursively finds friends and friends-of-friends using the KD-tree until no new friends can be added. This process yields quadratic runtime in dense regions, since every point is a friend of every other point. We obtain the conversion constants ρ_p, ρ_m (shown in Table 4.3) by executing 10 micro-benchmark runs of the analysis task over a 1% data sample.

4.3.1 Overall SkewReduce Performance

In this section, we present experimental results that answer the following question: **Q: Does SkewReduce improve task completion times in the presence of computational skew compared to uniform data partitioning? Is the improvement significant?**

To answer this question, we measure the total runtime of the plans generated by SkewReduce for both datasets. We compare them against the runtimes of a manually crafted plan called *Manual* and plans with various uniform partitioning granularities: *Coarse*, *Fine*, *Finer*, and *Finest*. All plans are generated from the same 1% data sample. *Coarse* mimics

Hadoop, which assigns a Map task to each HDFS chunk. Similarly, *Coarse* partitions the data into fragments that each contains the same number of data points. It does so by repeatedly splitting the region to bisect the data, one axis at a time in a round robin fashion, just like a KD-tree using a Recursive Coordinate Bisection (RCB) scheme [17]. *Coarse* stops splitting when the size of each partition is less than 128 MB. *Fine* stops splitting only when each partition is 16 MB. *Finer* and *Finest* partition the *Fine* partitions further until each partition holds 4 MB and 2 MB, respectively. Finally, we prepared the *Manual* plan by tweaking the *Fine* plan based on the execution results: we merged partitions experiencing no skew and split slow partitions further. We prepared a manual plan only for the Astro dataset due to the tedious nature of this task. Figure 4.3 shows the relative completion times of all plans compared to the optimized plan, labeled as *Opt*. We also report the actual completion time of each plan in the accompanying table.

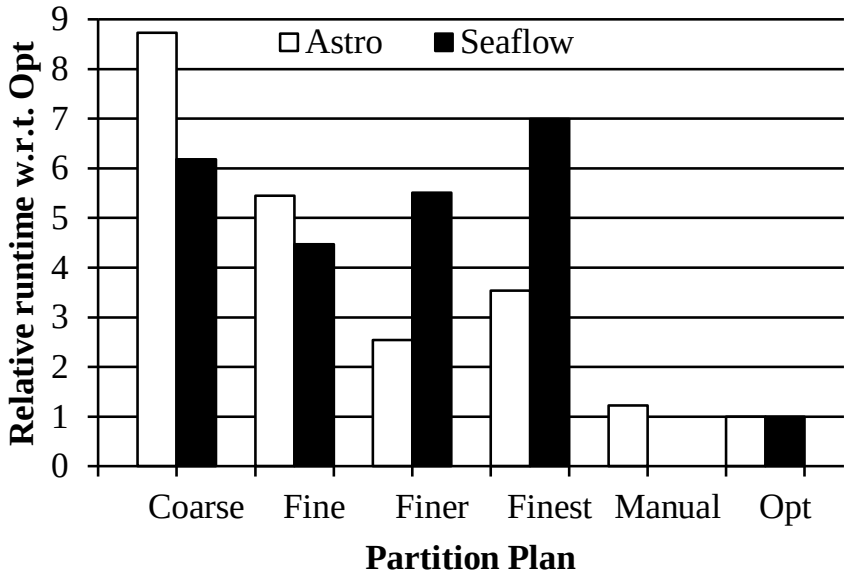
The results from both datasets illustrate that fine-grained uniform splitting only improves performance up to a certain point before runtimes increase again due to overheads associated with scheduling and executing so many partitions. The SkewReduce optimizer's plan, however, guided by user-defined cost functions, is more than twice as fast as the best uniform plan. For the Astro dataset, SkewReduce improves the completion time of the clustering task by a factor of more than 8 compared with *Coarse*, which is the strategy equivalent to the default approach in MapReduce-type systems. SkewReduce's performance is even a bit better than the Manual plan. For the Seaflow dataset, the *Opt* runtime is a factor of 3 better than *Fine* and a factor of 6 better than *Coarse*.

Overall, SkewReduce can thus significantly improve the runtime of this analysis task.

4.3.2 Cost Model Fidelity

In this section, we start to study the parameters that affect SkewReduce's performance. In particular, we answer the following question: **Q: How important is the fidelity of the cost model for SkewReduce's optimization?**

The fidelity of a cost function is related to the Fidelity property defined in Section 4.2.2. Given two partitions R and S, if a cost function C_A is more likely to produce cost estimates



Completion time (hours for Astro, minutes for Seaflow)

Dataset	**Coarse**	**Fine**	**Finer**	**Finest**	**Manual**	**Opt**
Astro	14.1	8.8	4.1	5.7	2.0	1.6
Seaflow	87.2	63.1	77.7	98.7	-	14.1

Figure 4.3: Relative runtime of different partitioning strategies compared with the optimized plan (Opt). The table shows the actual completion time for each strategy (units are hours for Astro and minutes for Seaflow). Manual plan is shown only for the Astro dataset. Overall, SkewReduce's optimization significantly improves the completion time.

that reflect the correct execution time order than a cost function C_B, we say that C_A is a higher-fidelity cost function than C_B.

To answer this question, we compare the performance of SkewReduce using different cost functions and find that plan quality is sensitive to the fidelity of the cost function to the actual algorithm. In this experiment, we compare three cost functions: the 3D histogram function described previously, a simpler but less faithful 1D histogram, and simply the data size as a cost proxy. Table 4.4 summarizes the three functions.

Function	Fidelity	Description
Data Size	Low	The number of data items
Histogram 1D	Medium	Sum of squared frequencies, 10 buckets
Histogram 3D	High	Sum of squared frequencies, all buckets

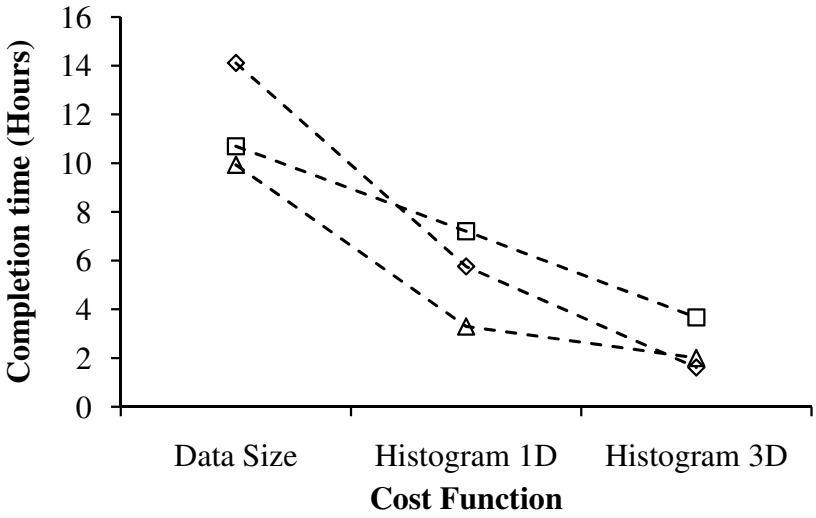
Table 4.4: Cost functions for evaluation

For each dataset, we prepared three independent 1% data samples. We then ran the SkewReduce optimizer with the different cost functions for each data sample and compared the execution times of the resulting partition plans.

Figure 4.4 shows the result for the Astro dataset. The x-axis shows the cost function in order of increasing fidelity. Each dashed line represents the execution time of plans generated from the same sample. Overall, the fidelity of the cost function significantly affects the total runtime.

The Data Size cost function leads to a plan essentially equivalent to *Coarse* in Figure 4.3 thus there is no big improvement. Interestingly, the *Histogram 1D* function yields a runtime close to the second best *Finest* from Figure 4.3. Hence, even a cost function with limited fidelity to the actual algorithm can help compare to not using SkewReduce. The most faithful *Histogram 3D* function yields the best plan and significantly improves the execution time compared to the least faithful cost function, Data Size. Across all three samples, the higher fidelity cost function yields the better plan. The estimated runtimes and percent errors with respect to the real execution times are shown in the accompanying table in Figure 4.4. The expected runtimes significantly deviate from the actual runtimes because of discrepancies between the cost function and the real algorithm, as well as the sample and the real data.

Figure 4.5 shows the results of the same experiment on the Seaflow dataset. Here, the Histogram 1D cost function yields a worse plan than the Data Size cost function in two out



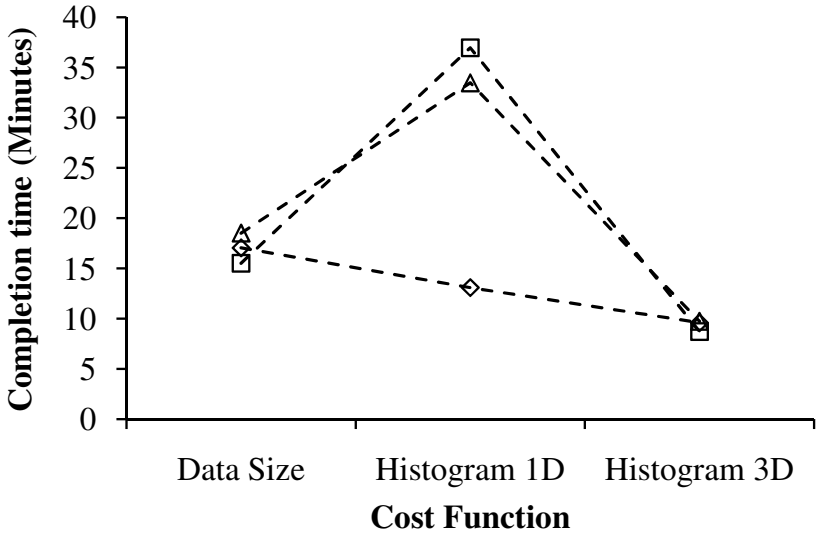
Estimated runtime and percent error for Astro data (hour)

Sample	Data Size	Histogram 1D	Histogram 3D
1	6.64 (-53.0%)	7.59 (31.8%)	2.84 (76.2%)
2	6.64 (-37.9%)	7.21 (8.20%)	2.94 (-20.0%)
3	6.64 (-33.2%)	3.30 (135%)	2.73 (36.2%)

Figure 4.4: Completion times of plans for the Astro dataset using different cost functions. The x-axis is cost functions in increasing order of fidelity and y-axis is completion time in hours. Each line represents the real runtimes of the plans derived from the same sample. The table shows estimated runtimes and percent errors with respect to completion times.

of three cases, while Histogram 3D consistently produces the best plans.

The anomaly is due to characteristics of the Seaflow dataset. Unlike the Astro dataset, all domains of the Seaflow dataset are 16-bit unsigned integers with significant repetition (*e.g.*, values near 0 along the x-axis in Figure 4.1). Histogram 1D tends to overestimate the cost of a partition compared with Histogram 3D because it approximates the cost using a fixed number of buckets (Table 4.4). With small domains and many repeated values in the dataset, the error becomes significant compromising fidelity and eventually affecting the optimization. While the resulting plans are worse than those produced by Data Size, the



Estimated runtime and percent error for Seaflow data (minutes)

Sample	Data Size	Histogram 1D	Histogram 3D
1	3.84 (-77.5%)	6.71 (-48.8%)	1.41 (-85.3%)
2	3.84 (-75.2%)	6.71 (-81.8%)	1.41 (-83.9%)
3	3.84 (-79.2%)	6.72 (-79.9%)	1.41 (-85.5%)

Figure 4.5: Completion time of plans for Seaflow dataset using different cost functions. Note that the y-axis is in minutes. Each line represents the real runtimes of the plans derived from the same sample. The table shows estimated runtimes and percent errors with respect to completion times. Histogram 1D performs badly because it significantly overestimates the cost of a partition.

execution time is only half of the best uniform partitioning strategy (*Fine*) in Figure 4.3. Thus, a less faithful cost function may not produce a good plan consistently but still yields better results than a uniform strategy.

Finally, the Data Size cost function significantly improves the runtime compared with the *Coarse* and *Fine* plans from Figure 4.3, while it seems that the two should be equivalent. The difference is attributable to SkewReduce's ability to select the partitioning axis that yields the best splits.

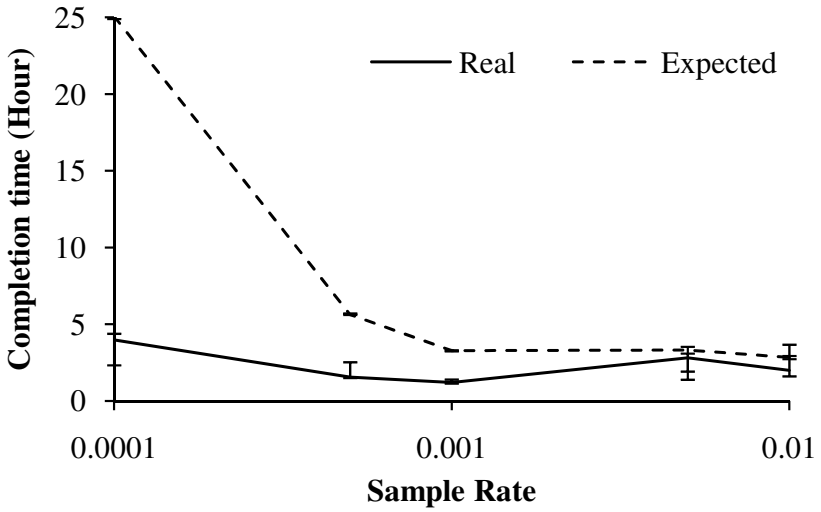


Figure 4.6: Completion time for the Astro dataset with varying sample rates. Error bars show the minimum and maximum values obtained for each sampling rate.

In summary, a high fidelity cost function benefits SkewReduce's optimization and improves the runtime significantly. However, even approximate cost functions can yield better plans than ignoring computation costs and splitting only based on dataset sizes.

4.3.3 Sample Size

In this section, we examine the effects of the sample size on SkewReduce's performance and answer the following question: **Q: What sample sizes are required for SkewReduce to generate good plans?**

SkewReduce's optimization is based solely on the sample, and an unrepresentative sample may affect the accuracy of the optimizer's cost estimates. To measure the effect on accuracy, we prepared three independent samples with varying sampling rates, then generated and executed an optimized plan using the best cost function, Histogram 3D.

Figures 4.6 and 4.7 show the results from the Astro and Seaflow datasets, respectively. In both figures, the optimizer's cost estimates improve as the sample size increases but the convergence is not smooth. Surprisingly, the estimated runtime of the Astro dataset

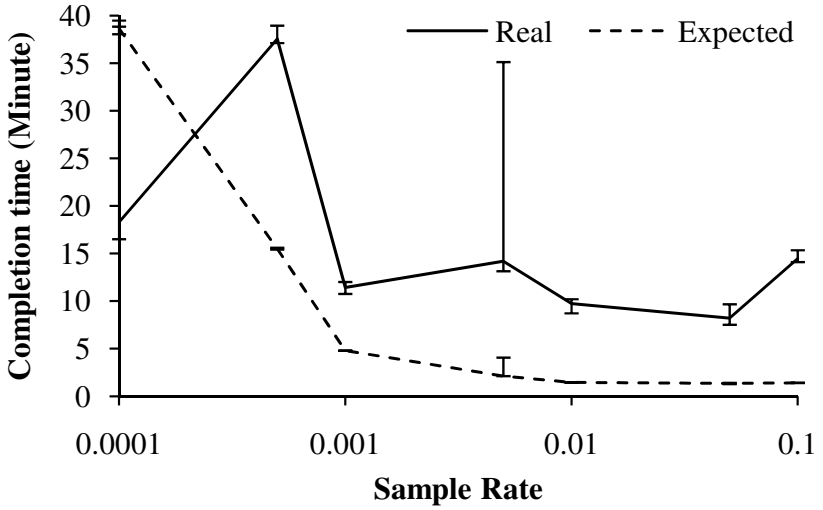


Figure 4.7: Completion time for the Seaflow dataset with varying sample rates. Error bars show the minimum and maximum values obtained for each sampling rate.

does not fluctuate as much as that of the Seaflow dataset even at lower sampling rates. The reason is that the extreme density variations in the Astro dataset that drive the performance are still captured even in a small sample. In contrast, the Seaflow sample may or may not exhibit significant skew. We also find that a larger sample does not always guarantee a better plan. In Figure 4.7, the sampling rate of 10% does not yield a better plan than a 5% sampling rate. The conclusion is that the quality of optimization may vary subject to the representativeness of the sample. Interestingly, the runtime of this suboptimal plan is still a factor of 2 improvement compared to the plans based on uniform partitioning as shown in Figure 4.3.

4.3.4 SkewReduce Overhead

We finally study SkewReduce's overhead and answer the following question. **Q:** *How long does SkewReduce's optimization take compared with the time to process the query?*

Figure 4.8 shows the runtime of the prototype optimizer using the Data Size and the Histogram 3D cost functions for each dataset. At a 1% sampling rate, the optimization

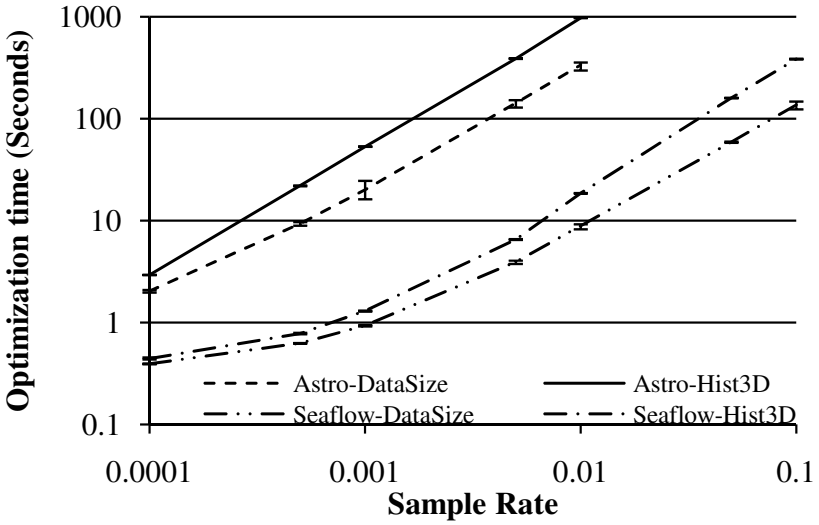


Figure 4.8: Optimization time with varying sample rates and cost functions. With a 0.01 sample rate, there are 590K samples for the Seaflow dataset and 9.1M samples for Astro.

takes 18 seconds using 594K samples from the Seaflow dataset and 15 minutes using 9.1 M samples from the Astro dataset. Considering that the prototype is not parallelized and does not manage memory in any sophisticated way, the runtime is still a small fraction of the actual runtime of the algorithm for each dataset. With an efficient parallel implementation, the SkewReduce optimizer could potentially run with a more complex cost function or use multiple samples to produce a better plan.

4.3.5 Data Volume between Operations

Lastly, we briefly consider the performance implications of SkewReduce's capability to set-aside some data during the process and merge steps. The amount of data transferred between stages is known to be a bottleneck of MapReduce [43]. In SkewReduce, however, this is more than a bottleneck because a flood of data between levels eventually exceeds the memory bounds of merge. In this section, we answer the following question. **Q:** *Does SkewReduce feed too much data to merge? How much data is set aside?*

In Figure 4.9, we analyze the total amount of data generated at each level of the partition

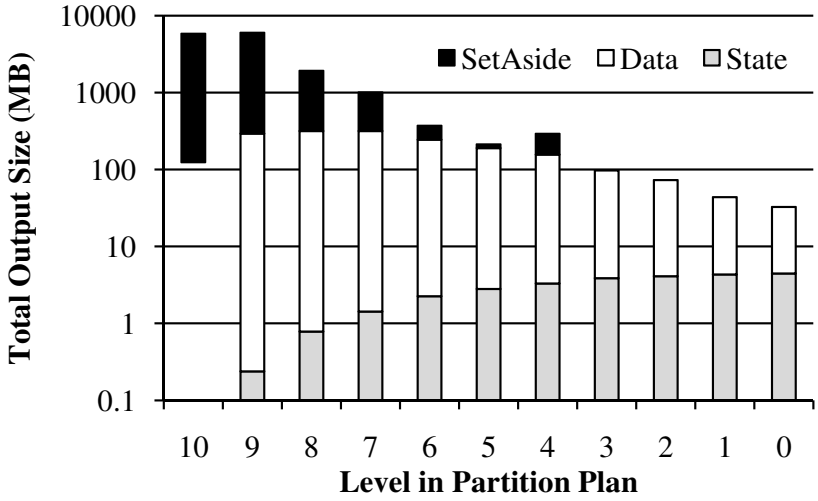


Figure 4.9: Aggregate output data size produced at each level of an optimized partition plan for the Astro dataset. The x-axis is the level in the partition tree with level 10 representing the leaves. The y-axis is in log scale. Data and State together form the Feature object (F). Data corresponds to boundary particles with cluster ids and State holds the cluster mappings found so far. Overall, a significant amount of data is set-aside and only a small amount of data is passed to merge.

tree during one execution of the clustering algorithm on the Astro dataset. Overall, a total of 13.7 GB of data was set aside by process, which corresponds to almost 99% of the input data. The total data passed to all merge operators is only 2 GB and the top-level merge has received the most amount of data (39 MB), however, this is only one third of the 128 MB chunk size. Thus, through its API, SkewReduce guides application writers toward an efficient implementation of their feature extraction applications, setting aside significant amounts of data when possible, reducing traffic between operators, and helping to satisfy the per node memory requirement. We acknowledge that the amount of data that is set aside data will vary depending on the dataset and the application.

4.4 Conclusion

In this chapter, we presented SkewReduce, a new API for feature-extracting scientific applications and an implementation that leads to an efficient execution of these applications. At the heart of SkewReduce is a static optimizer that leverages user-supplied cost functions to generate parallel processing plans, which significantly reduce the impact of computational skew inherent in many of these applications. Through experiments on two real datasets, we showed that SkewReduce can improve application run times by a factor of two to eight and takes only seconds to minutes to run, facilitating offline resource planning.

Chapter 5

SKEWTUNE: DYNAMIC SKEW MITIGATION

The SkewReduce approach to skew avoidance presented in Chapter 4 can dramatically improve job completion times by optimizing how data is partitioned across tasks based on user-defined cost models. Although SkewReduce can significantly reduce the impact of skew, it has several limitations due to its static nature. First, SkewReduce can not handle runtime conditions such as failures and interferences. Second, estimation errors from sample and user-defined cost models may underestimate or overestimate the cost of a partition, so the optimized partition plan may still exhibit skew, though to a much lesser degree than before the optimization. Third, SkewReduce burdens the user with specifying cost functions. Finally, SkewReduce is specific to one type of applications.

In this chapter, we propose SkewTune, a dynamic skew mitigation strategy for the parallel evaluation of UDOs. SkewTune addresses several of the limitations of SkewReduce. The greatest benefit of SkewTune is its transparency. Users can use SkewTune with existing MapReduce applications without modifications. The output of a mitigated job is identical to the output without mitigation; thus, mitigation does not break downstream applications that consume the output of a mitigated job. SkewTune detects a task experiencing skew (*i.e.*, a *straggler task* as defined in Section 3.3.2) by monitoring the execution of each operator partition. It thus does not require cost models from users. If a straggler task is detected, SkewTune mitigates skew at runtime by dynamically repartitioning the input data of the straggler task (*i.e.*, it *tunes* the impact of skew at runtime by repartitioning the input to the straggler task). The data repartitioning process preserves the order of the input data, so the final output after mitigation can be transparently reconstructed by concatenating the output of the extra tasks resulting from skew mitigation. SkewTune is independent of SkewReduce, but it can complement SkewReduce by dynamically mitigating skew resulting from estimation errors during the static partition optimization and by handling unexpected

failures and load variations. These benefits come at one expense: a UDO should not maintain state across invocations (*i.e.*, it should process each input record independently of the others).

SkewTune effectively handles two very common types of skew: (1) skew caused by an uneven distribution of input data to operator partitions (or tasks) and (2) skew caused by some portions of the input data taking longer to process than others. As shown in Section 3.3.4, for these sources of skew, speculative execution, a popular strategy in MapReduce-like systems [37, 61, 68] to mitigate skew stemming from a non-uniform performance of physical machines, is ineffective because the speculative tasks execute the same code on the same data and therefore do not complete in any less time than the original tasks. SkewReduce could help by identifying expensive tasks and splitting them through the partition optimization, but, as mentioned above, it is insufficient because of estimation errors and dynamically changing runtime conditions that are not taken into account during the optimization. Additionally, SkewReduce requires extra inputs from the user, while SkewTune does not.

SkewTune is designed for MapReduce-type engines, characterized by disk-based processing and a record-oriented data model. The technique is applicable to other parallel data flow engines with an appropriate isolation layer that can replay the output of each operator partition on a request from the downstream operator (*e.g.*, [134]). We implemented the SkewTune technique by extending the Hadoop MapReduce engine [61]. SkewTune relies on two properties of the MapReduce model: (1) MapReduce's ability to buffer the output of an operator before transmitting it to the next operator; and (2) operator de-coupling, where each operator processes data as fast as possible without back-pressure from downstream operators. SkewTune's optimizations mitigate skew while preserving the fault-tolerance and scalability of vanilla MapReduce.

The key features of SkewTune are:

- SkewTune mitigates two very common types of skew: skew due to an uneven distribution of data to operator partitions and skew that results from the fact that some subsets of the data take longer to process than others.
- SkewTune can optimize unmodified MapReduce programs; programmers need not

change a single line of code.

- SkewTune preserves interoperability with other UDOs. It guarantees that the output of an operator consists of the same number of partitions with data sorted in the same order within each partition as an execution without SkewTune.
- SkewTune is compatible with pipelining optimizations proposed in the literature (*c.f.*, [134]) and does not require any synchronization barrier between consecutive operators[1].

We evaluate SkewTune through experiments with real data and real applications including PageRank [25], CloudBurst [116], and an application that builds an inverted index over Wikipedia. We show that SkewTune can reduce processing times by up to factor of 4 when skew arises and adds only minimal overhead in the absence of skew. Most importantly, SkewTune delivers *consistent* performance independent of the initial configuration of a MapReduce job.

The rest of this chapter is organized as follows. We discuss the design requirements of SkewTune in Section 5.1. We present the SkewTune approach in Section 5.2 and the Hadoop implementation in Section 5.3. We show results from experiments with real application in Section 5.4.

5.1 SkewTune Design Requirements

Before presenting the SkewTune approach, we first discuss the rationale behind its design. When designing SkewTune, we had the following goals in mind:

Developer Transparency. The first goal behind SkewTune is to make it easier for MapReduce developers to achieve high performance. For this reason, we do not want these developers to even be aware that skew problems can arise. We want SkewTune to simply be an improved version of Hadoop that executes their jobs faster. As a result, we reject all design alternatives that require operator writers to either implement their jobs following special templates [14] or provide special inputs such as cost functions for their operators [81].

[1]However, SkewTune, like MapReduce, does not allow downstream operators to throttle the flow of upstream operators, as is typically the case in parallel pipelined query plans.

Instead, SkewTune should operate on unchanged MapReduce jobs.

Mitigation Transparency. Today, MapReduce makes certain guarantees to users: The output of a MapReduce job is a series of files, with one file per reducer. The user can configure the number of reducers. Additionally, the input of each reducer is sorted on the reduce key by the user-provided comparator function thus the output is produced in a specific order. To facilitate adoption and to ensure the correctness and efficiency of the overall application, we want SkewTune to preserve these guarantees. The output of a job executed with SkewTune should be the same as the output of a job executed without SkewTune: it should include the same number of files with the same data order inside these files. Indeed, users often create data analysis workflows and the application consuming the output of a MapReduce job may rely on there being a specific number of files and on the data being sorted within these files. By preserving these properties, SkewTune also helps ensure predictability: the same job executed on the same input data will produce the same output files in the same order.

Maximal Applicability. In MapReduce (and in other parallel data processing systems), many factors can cause skew in a UDO as surveyed in Section 3.2. We designed SkewTune to handle these different types of skew rather than specializing SkewTune for only one type of skew [37, 67]. In general, SkewTune strives to make the least number of assumptions about the cause of skew. Instead, it monitors execution, notices when some tasks run slower than others, and reacts accordingly independent of the reason why the tasks are slower.

No Synchronization Barriers. Finally, parallel data processing systems try to minimize global synchronization barriers to ensure high performance [80] and produce incremental results when possible. Even in MapReduce, reducers are allowed to start copying data before the previous mappers finish execution. Additionally, new MapReduce extensions strive to further facilitate pipelining during execution [86, 31, 134]. For those reasons, we avoided any design options that required blocking while an operator finishes processing before letting the next operator begin shuffling (and possibly processing) the data.

To achieve the above goals, SkewTune only assumes that a MapReduce job follows the API contract: each `map()` and `reduce()` invocation is independent. This assumption enables SkewTune to automate skew mitigation because it can be sure that re-partitioning

input data at the boundary of map and reduce function invocations is safe. Such re-partitioning will not break the application logic.

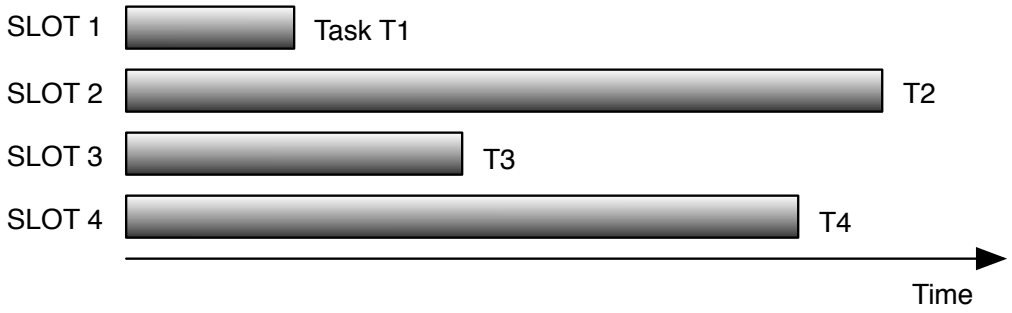
5.2 SkewTune Approach

SkewTune is designed to be API-compatible with Hadoop, providing the same parallel job execution environment while adding capabilities for detecting and mitigating skew. This section presents SkewTune's approach and core algorithms; Section 5.3 describes the implementation on top of Hadoop.

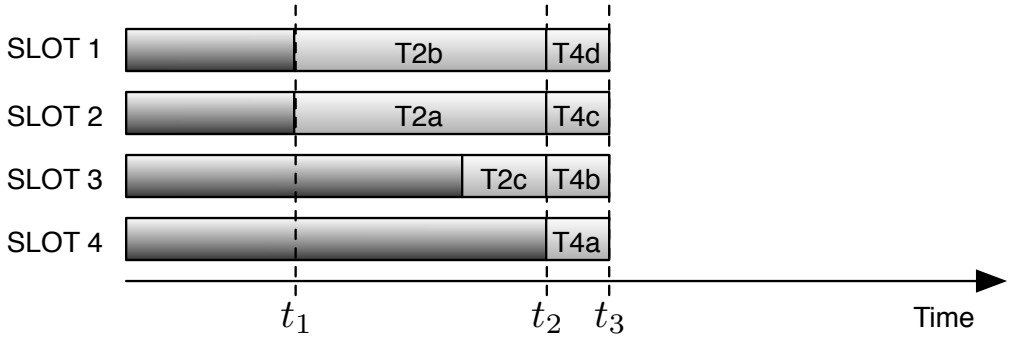
5.2.1 Overview

SkewTune takes a Hadoop job as input. For the purpose of skew mitigation, SkewTune considers the map and reduce phases of the job as separate UDOs. In SkewTune, as in Hadoop, a UDO pulls its input from the output of the previous UDO, where it is buffered locally. A UDO is assumed to take a *record* as input. A key-value pair (*i.e.*, mapper input) and a key group (*i.e.*, reducer input) are each considered a special case of a record. Each UDO is parallelized into tasks, and each task is assigned a *slot* in the cluster. There is typically one slot per CPU core per node. When a task completes, the slot becomes available.

SkewTune's skew mitigation technique is designed for MapReduce-type data processing engines. The three important characteristics of these engines with respect to skew handling are the following: (1) A coordinator-worker architecture where the coordinator node makes scheduling decisions and worker nodes run their assigned tasks. On completion of a task, the worker node requests a new task from the coordinator. This architecture is commonly used today [37, 43, 61, 68]. (2) De-coupled execution: Operators do not impose back-pressure on upstream operators. Instead, they execute independently of each other. (3) Independent record processing: The tasks are executing a UDO that processes each input record (possibly nested) independently of each other. Additionally, SkewTune requires (4) Per-task progress estimation, t_{remain}, which estimates the time remaining [98, 148] for each task. Each worker periodically reports this estimate to the coordinator. (5) Per-task statistics: each task keeps



(a) Without SkewTune, operator runtime is that of the slowest task.



(b) With SkewTune, the system detects available resources as task T1 completes at t_1. SkewTune identifies task T2 as the straggler and re-partitions its unprocessed input data. SkewTune repeats the process until all tasks complete.

Figure 5.1: Conceptual skew mitigation in SkewTune

track of a few basic statistics such as the total number of (un)processed bytes and records.

Figure 5.1 illustrates the conceptual skew mitigation strategy of SkewTune. Without SkewTune, the operator completion time is dominated by the slowest task (*e.g.*, T2 in Figure 5.1a). With SkewTune, as shown in Figure 5.1b, the system detects that T2 is experiencing skew at t_1 when T1 completes. SkewTune labels T2 as *the straggler* and mitigates the skew by repartitioning T2's *remaining* unprocessed input data. Indeed, T2 is not killed but rather terminates early as if all the data that it already processed was the only data it was allocated to process. Instead of repartitioning T2's remaining input data across only slots 1 and 2, SkewTune *proactively* repartitions the data to also exploit slot 3, which is expected to become available when T3 completes. SkewTune re-partitions the data such

$\mathcal{N}$	Set of nodes in the cluster
$\mathcal{S}$	Set of slots in the cluster (multiple slots per node)
$\mathcal{O}$	Set of output files
$\mathcal{R}$	Set of running tasks
$\mathcal{W}$	Set of unscheduled tasks
Δ	Straggler's unprocessed data (bytes)
β	Disk bandwidth (bytes/seconds)
ρ	Task scheduling overhead (seconds)
ω	Repartitioning overhead (seconds)
T, t_{remain}	A task and its time-remaining (seconds)

Table 5.1: Notations in Section 5.2

that all new partitions are predicted to complete at the same time. The resulting subtasks T2a, T2b, and T2c are called *mitigators* and are scheduled in the longest processing-time first manner. SkewTune repeats the detection-mitigation cycle until all tasks complete. In particular, at time t_2, SkewTune identifies T4 as the next straggler and mitigates the skew by repartitioning T4's remaining input data.

In terms of our requirements from Section 5.1, SkewTune achieves *developer transparency* by detecting and mitigating skew at runtime without requiring any input from the developers. We further discuss SkewTune's skew detection approach in Section 5.2.2. To achieve *mitigation transparency*, SkewTune re-partitions the straggler's data using range-partitioning as we discuss further in Section 5.2.3. To be *maximally applicable*, SkewTune makes no assumptions about the cause of the skew. It also respects the input record boundary when repartitioning data. Thus, as long as a UDO follows the MapReduce API contract, SkewTune is applicable without breaking the application semantics. Finally, SkewTune's skew mitigation approach does not require any synchronization barriers.

5.2.2 Skew Detection

Skew detection determines *when* to mitigate skew experienced by *which* task. If the detection is too eager, SkewTune may split a task and pay unnecessary overhead (*i.e.*, false-positive). If the detection is too conservative, SkewTune may miss the right mitigation timing thus diminishing the skew-mitigation gains (*i.e.*, false-negative).

Late Skew Detection: SkewTune's skew detection approach relies on the fact that tasks in consecutive phases are decoupled from each other. That is, map tasks can process their input and produce their output as fast as possible. They never block waiting for reduce tasks to consume that data. Similarly, reduce tasks can never be blocked by map tasks in a subsequent job.

This decoupling has important implications for skew handling. Because tasks can independently process their input as fast as possible, the cluster has high utilization as long as each slot is running some task. For this reason, SkewTune delays any skew mitigation decisions until a slot becomes available. We call this approach *late skew detection*. Late skew detection is analogous to MapReduce's current speculative execution mechanism [37, 61], where slow remaining tasks are replicated when slots become available. Similarly, SkewTune's repartitioning overhead is only incurred when there are idle resources. Late skew detection thus reduces opportunities for false positives. At the same time, it avoids false negatives by immediately allocating resources when they become available.

Identifying Stragglers: The next key question is to decide which task to label as the straggler. Here, we observe that it is never beneficial to re-partition more than one task at a time, since re-partitioning one task can suffice to fully occupy the cluster again. Given that only one task should be labeled as a straggler, SkewTune selects the task with the greatest t_{remain} estimate at the time of detection.

SkewTune flags skew when half of the time remaining is greater than the repartitioning overhead:

$$\frac{t_{remain}}{2} > \omega$$

The intuition is as follows. If SkewTune decides to repartition task T, at least two slots become available: the slot running T and the slot that recently became idle and triggered skew

Algorithm 5 GetNextTask()

Input: $\mathcal{R}$: set of running tasks

$\mathcal{W}$: set of unscheduled waiting tasks

inProgress: global flag indicating mitigation in progress

Output: a task to schedule

```
1: task ← null
2: if W ≠ ∅ then
3:    task ← chooseNextTask(W)
4: else if ¬inProgress then
5:    task ← argmax_{task∈R} time_remain(task)
6:    if task ≠ null ∧ time_remain(task) > 2 · ω then
7:       stopAndMitigate(task)  // asynchronous
8:       task ← null
9:       inProgress ← true
10:   end if
11: end if
12: return task
```

detection. After paying repartition overhead ω, the expected remaining time would be half of the remaining time of T (Table 5.1 summarizes the notation). The repartition thus only makes sense if the original runtime of T is greater than the new runtime plus the overhead. In our prototype implementation, ω is on the order of 30 seconds (see Section 5.4). Hence, our prototype only re-partitions tasks if at least 1 minute worth of processing remains. For long-running tasks where skew is particularly damaging, overhead of a few minutes is typically negligible.

Algorithm 5 summarizes SkewTune's skew detection strategy. As long as there exist unscheduled tasks, SkewTune invokes the ordinary task scheduler `chooseNextTask()`. If the coordinator runs out of tasks to schedule, SkewTune starts to consider repartitioning one of the running tasks based on the t_{remain} estimates. `stopAndMitigate()` asynchronously notifies the chosen task to stop and to commit the output produced so far. We describe the mitigation process next.

5.2.3 Skew Mitigation

There are three challenges related to mitigating skew through repartitioning. First, we want to minimize the number of times that we repartition any task to reduce repartitioning overhead. Second, when we repartition a straggler, we want to minimize any visible side-effects of the repartitioning to achieve mitigation transparency (see Section 5.1). Finally, we want to minimize the total overhead of skew mitigation, including any unnecessary recomputations.

SkewTune strives to minimize the number of repartition operations by identifying one straggler at a time and proactively partitioning its data in a manner that accounts for slots that are likely to become available in the near future. To eliminate side-effects of skew mitigation, SkewTune uses range partitioning to ensure that the original output order of the UDO result is preserved. To minimize the mitigation overhead, SkewTune saves a straggler's output and repartitions only its unprocessed input data. It also uses an inexpensive, linear-time heuristic algorithm to plan mitigators. To drive this planning, SkewTune needs to collect information about the value distribution in the repartitioned data. To minimize overhead, SkewTune makes a cost-based decision to scan the remaining data locally at the straggler or to spawn new tasks that scan the distributed input in parallel.

Skew mitigation occurs in three steps. First, the straggler stops its computation. Second, depending on the size of the data that remains to be processed, either the straggler or the operators upstream from the straggler collect statistics about the straggler's remaining input data. Finally, the coordinator plans how to re-partition the straggler's remaining work and schedules the mitigators. We now present these steps in more detail.

Stopping a Straggler

When the coordinator asks a straggler to stop, the straggler captures the position of its last processed input record, allowing mitigators to skip previously processed input. If the straggler is in a state that is impossible or difficult to stop (*e.g.*, processing the last input record or performing the local sort at the end of the map phase), the request fails and the coordinator either selects another straggler or repartitions and reprocesses the entire

straggler's input if this straggler is the last task in the job. Reprocessing a straggler's entire input is analogous to MapReduce's speculative execution [37, 61] except that SkewTune repartitions the input before reprocessing it.

Scanning Remaining Input Data

In order to ensure skew mitigation transparency, SkewTune uses range-partitioning to allocate work to mitigators. With this approach, the data order remains unchanged between the original MapReduce job and the altered job. The output of the mitigators only needs to be concatenated to produce an output identical to the one obtained without SkewTune. An alternate design would be to use hash-partitioning and add an extra MapReduce job to sort-merge the output of the mitigators. Such an extra job would add overhead. Additionally, a hash function is not guaranteed to evenly balance load between mitigators, especially if the number of keys happens to be small. Range partitioning avoids both problems.

When range-partitioning data, a data range for a map task takes the form of an input file fragment (*i.e.*, file name, offset, and length). A range for a reduce task is an interval of reduce keys. In the rest of this section, we focus on the case of repartitioning the reduce task's input. The techniques are equally applicable to map tasks.

Range-partitioning a straggler's remaining input data requires information about the content of that data: The coordinator needs to know the key values that occur at various points in the data. SkewTune collects that information before planning the mitigator tasks.

A naïve approach is to scan the data and extract all keys together with the associated record sizes. The problem with this approach is that it may produce a large amount of data if there exists a large number of distinct keys. Such large data imposes a significant network overhead and also slows-down the mitigator planning step.

Instead, SkewTune collects a compressed summary of the input data. The summary takes the form of a series of key intervals. Each interval is approximately the same size in bytes, respecting the input boundaries (*e.g.*, a single record for map, values sharing a common reduce key for reduce). These intervals become the units of range-partitioning. Consecutive intervals can be merged to create the actual data range assigned to a mitigator.

Choosing the Interval Size: Given $|\mathcal{S}|$, the total number of slots in the cluster, and Δ, the number of unprocessed bytes, SkewTune needs to generate at least $|\mathcal{S}|$ intervals since it is possible that all cluster slots will be available for mitigators. However, because SkewTune may want to allocate an uneven amount of work to the different mitigators (*e.g.*, Figure 5.1), SkewTune generates $k|\mathcal{S}|$ intervals. Larger values of k enable finer-grained data allocation to mitigators but they also increase overhead by increasing the number of intervals and thus the size of the data summary. In our prototype implementation, k is set to 10. Hence, the size s of the intervals is given by $s = \lfloor \frac{\Delta}{k \cdot |\mathcal{S}|} \rfloor$.

Local Scan: If the size of the remaining straggler data is small, the worker running the straggler scans that data and generates the intervals. Algorithm 6 summarizes the interval generation process. The algorithm expects a stream of intervals I as input. This is the stream of *singleton intervals*, with one interval per key in the reducer's input. For the local scan, b is set to s and k is ignored. The algorithm iterates over these singleton intervals. To generate the output intervals, it opens an interval with the first seen key. It then merges the subsequent keys and their statistics (*e.g.*, size of all values in bytes) until the aggregated byte size reaches the threshold s. If a key has a byte size larger than s, the key remains in its own singleton interval. The process continues until the end of the data.

Choosing between a Local and a Parallel Scan: To choose between a local and a parallel scan, SkewTune compares the estimated cost (in terms of total time) for each approach. The time for the local scan is given by $\frac{\Delta}{\beta}$, where Δ is the remaining input data in bytes and β is the local disk bandwidth. The time for the parallel scan is the time to schedule an extra MapReduce job to perform the scan, and the time for that job to complete. The latter is equal to the time that the slowest task in the job, say n, will take to scan its input data: $\frac{\sum_{o \in O_n} o.bytes}{\beta}$, where O_n is the set of all map outputs at node n (recall that multiple map tasks can run on a node). The decision is thus made by testing the following inequality:

$$\frac{\Delta}{\beta} > \frac{\max\{\sum_{o \in O_n} o.bytes \mid n \in \mathcal{N}\}}{\beta} + \rho$$

where $\mathcal{N}$ is the set of nodes in the cluster and ρ is the task scheduling delay. The stopping

Algorithm 6 GenerateIntervals()

Input: I: Sorted stream of intervals

b: Initial bytes-per-interval. Set to s for local scan.

s: Target bytes-per-interval.

k: Minimum number of intervals.

Output: list of intervals

```
1: result ← []  // resulting intervals
2: cur ← new_interval()  // current interval
3: for all i ∈ I do
4:   if i.bytes > b ∨ cur.bytes ≥ b then
5:     if b < s then
6:       result.appendIfNotEmpty(cur)
7:       if |result| ≥ 2 × k then
8:         // accumulated enough intervals. increase b.
9:         b ← min{2 × b, s}
10:        // recursively recompute buffered intervals
11:        result ← GenerateIntervals(result, b, b, k)
12:      end if
13:    else
14:      result.appendIfNotEmpty(cur)
15:    end if
16:    cur ← i  // open a new interval
17:  else
18:    cur.updateStat(i)  // aggregate statistics
19:    cur.end ← i.end
20:  end if
21: end for
22: result.appendIfNotEmpty(cur)
23: return result
```

straggler tests the inequality since it knows where its input data came from. If a parallel scan is expected to be more cost-effective, the straggler immediately replies to the coordinator

Interval ID	Begin key	# values	End key
i_1	$k_3 : 4$	9	$k_7 : 3$
i_2	$k_7 : 1$	10	$k_{100} : 2$
i_3	$k_{50} : 2$	14	$k_{95} : 5$

Input: Intervals from Parallel Local Scans.

Key Range	Est. # values	Intervals
$[k_3, k_3]$	4	i_1
(k_3, k_7)	9	i_1
$[k_7, k_7]$	4	i_1, i_2
(k_7, k_{50})	$10/5$	i_2
$[k_{50}, k_{50}]$	$2 + 10/5$	i_2, i_3
(k_{50}, k_{95})	$14 + 10/5$	i_2, i_3
$[k_{95}, k_{95}]$	$5 + 10/5$	i_2, i_3
(k_{95}, k_{100})	$10/5$	i_2
$[k_{100}, k_{100}]$	2	i_2

Merge intervals and estimate # of values. The # values of i_2 (10) is evenly distributed over (k_7, k_{100}) range.

$\rightarrow$

Key Range	Est. # values
$[k_3, k_3]$	4
(k_3, k_7)	9
$[k_7, k_7]$	4
(k_7, k_{50})	2
$[k_{50}, k_{50}]$	4
(k_{50}, k_{95})	16
$[k_{95}, k_{95}]$	7
(k_{95}, k_{100})	2
$[k_{100}, k_{100}]$	2

Output: Aligned key ranges and estimated # of values.

Figure 5.2: Merging Result of Parallel Scan. The table on the left shows the output of the parallel scan. The middle column *#values* represents the number of values that fall between *begin* and *end* keys. Each key is also associated with its number of values (the number followed by ':'). The table on the right shows the output from merging the input intervals and the estimated number of values for each range. The values of wide interval (k_7, k_{100}) introduce uncertainty. The middle table shows how the 10 values of i_2 are evenly redistributed across the five key ranges included in (k_7, k_{100}).

and the latter schedules the parallel scan.

Parallel Scan: During a parallel scan, Algorithm 6 runs in parallel over the distributed input data (*i.e.*, map outputs). The intervals generated for each map output file are then put together to estimate the intervals that would have been generated by a local scan (illustrated in Figure 5.2).

The s value for the Local Scan may be too large for a parallel scan because there are usually more map outputs than the total number of slots in the cluster. Thus, we set a smaller s value for the parallel scan to properly generate intervals for each map output:

$$s = \lfloor \frac{\Delta}{k \cdot \max\{|\mathcal{S}|, |\mathcal{O}|\}} \rfloor$$

where $\mathcal{O}$ is the union of all the O_n sets. Additionally, because the size of the map output files can be skewed and because SkewTune does not know how much data in each of these files will have to be re-processed, SkewTune dynamically adjusts the interval size (variable b in Algorithm 6) starting from a small value (*e.g.*, 4 KB in prototype) and adaptively increasing it as it sees more unprocessed data. Whenever the b value is doubled, the collected intervals so far are merged using the new b value (line 7-12). Once the b value becomes s, the algorithm reaches a steady state and produces intervals every s bytes. Without this approach, a single wide key-interval may be generated for small data files and such wide key-intervals yield errors during the interval merge process at the coordinator.

Merging Intervals from a Parallel Scan: The intervals generated by a parallel scan are put together to approximate the result of a local scan. We present a two-pass algorithm illustrated in Figure 5.2, which estimates the number of records that would have been output by a local scan. The algorithm can be extended to handle other measures (*e.g.*, the number of bytes and the number of keys) in a straightforward manner.

The leftmost table in Figure 5.2 shows the input of the algorithm, which is a list of possibly overlapping intervals. Each interval is represented with a triple (begin key, # values, end key). The # *values* field represents the number of values that fall between *begin* and *end* keys. Each key is also associated with its number of values. The input is sorted by the *begin* key of each interval.

Intervals are a lossy representation of the data distribution. As a simplification, our

approach assumes that values are uniformly distributed in each interval. A small complication arises when intervals overlap. For example, i_3 is completely included in i_2 and it is uncertain how the 10 values of i_2 are actually distributed in the (k_7, k_{100}) range due to i_3. The middle table of Figure 5.2 shows addressing such uncertainty by evenly distributing the values over (k_7, k_{100}) range given the input. In the input, we can be sure that k_{50} and k_{95} exist between k_7 and k_{100}. The two keys segment the uncertain range (k_7, k_{100}) into five intermediate ranges: $(k_7, k_{50}), [k_{50}, k_{50}], (k_{50}, k_{95}), [k_{95}, k_{95}]$, and (k_{95}, k_{100}). Then we evenly distribute the 10 values of i_2 over the five ranges. For this approximation, thus we need to know how many keys fall within (or overlap) each of input interval.

The algorithm proceeds in two passes over the input data. During the first pass, the algorithm collects statistics about existing intervals and how they overlap. It then generates the smaller key ranges and estimates the number of values in these smaller ranges during the second pass. Figure 5.2 shows an example of input and output for this algorithm. We omit the algorithm pseudocode since it is straightforward.

The time and space complexities of the algorithm are $O(|I| \log |I|)$ and $O(|I|)$ respectively where I is the list of input intervals. The size of I is controlled by value k in Algorithm 6 during the parallel scan. If $|I|$ is expected to be too large to fit in memory, the k value needs to be adjusted to a smaller value. The output of the algorithm can also be post-processed by Algorithm 6 (*i.e.*, merge small adjacent key ranges to make the final intervals roughly the size of s).

Planning Mitigators

Finally, we present SkewTune's approach to planning mitigators. The goal is to find a contiguous order-preserving assignment of intervals to mitigators, meaning that the intervals assigned to a mitigator should be totally ordered on the key and should be contiguous: *i.e.*, no intervals between the first and the last keys should be assigned to other mitigators. The assignment should also minimize the completion time of all re-allocated data.

The planning algorithm should be fast because it is on the critical path of the mitigation process. A longer execution time means a longer idle time for the available slot in the

Algorithm 7 LinearGreedyPlan()

Input: I: a sorted array of intervals
T: a sorted array of t_{remain} for all slots in the cluster
θ: time remaining estimator
ω: repartitioning overhead
ρ: task scheduling overhead

Output: list of intervals

// Phase 1: find optimal completion time *opt*.
$opt \leftarrow 0;\ n \leftarrow 0$ // n: # of slots that yield optimal time
$W \leftarrow \theta(R)$ // remaining work+work running in n nodes
// use increasingly many slots to do the remaining work
while $n < |T| \wedge opt \geq T[n]$ **do**
  $opt' \leftarrow \frac{W+T[n]+\rho}{n+1}$ // optimal time using $n+1$ slots
  if $opt' - T[n] < 2 \cdot \omega$ **then**
    break // assigned too little work to the last slot
  end if
  $opt \leftarrow opt';\ W \leftarrow W + T[n] + \rho;\ n \leftarrow n + 1$
end while
// Phase 2: greedily assign intervals to the slots.
$P \leftarrow []$ // intervals assigned to slots
$end \leftarrow 0$ // index of interval to consider
while $end < |I|$ **do**
  $begin \leftarrow end;\ remain \leftarrow opt - T[|P|] - \rho$
  while $remain > 0$ **do**
    $t_{est} \leftarrow \theta(I[end])$ // estimated proc. time of interval
    if $remain < 0.5 \cdot t_{est}$ **then**
      break // assign to the next slot
    end if
    $end \leftarrow end + 1;\ remain \leftarrow remain - t_{est}$
  end while
  if $begin = end$ **then**
    $end \leftarrow end + 1$ // assign a single interval
  end if
  P.`append`(`new_interval`($I[begin], I[end-1]$))
end while
return P

cluster. We now describe a heuristic algorithm with linear time complexity with respect to the number of intervals.

Algorithm 7 takes as input the time remaining estimates for all active tasks in the cluster, the intervals collected by the data scan, a time remaining estimator θ, which serves to estimate processing times for intervals from their statistics (*e.g.*, sizes in bytes), and overhead parameters. The algorithm proceeds in two phases. The first phase (line 1-10) computes the optimal completion time *opt* assuming a perfect split of the remaining work (*i.e.*, record boundaries are not honored). The phase stops when a slot is assigned less than 2ω work to avoid generating arbitrarily small mitigators (line 6-7). 2ω is the largest amount of work such that further repartitioning is not beneficial. In the second phase, the algorithm sequentially packs the intervals for the earliest available mitigator as close as possible to the *opt* value. The algorithm then repeats the process for the next available mitigator until it assigns all the intervals to mitigators. The time complexity of this algorithm is $O(|I| + |\mathcal{S}| \log |\mathcal{S}|)$ where $|I|$ is the number of intervals and $\mathcal{S}$ is the number of slots in the cluster.

5.2.4 Discussion

SkewTune in a Shared Cluster: SkewTune currently assumes that a single user has access to all the resources in a cluster. There are two ways to incorporate SkewTune in a shared cluster setup: (1) by using a task scheduler that carves out a pre-defined set of resources for each user or (2) by implementing a SkewTune-aware scheduler that prioritizes mitigators (and preempts other tasks if necessary) if mitigating a straggler improves overall cluster utilization and latency.

Very expensive map() or reduce(): SkewTune is designed to repartition load around record boundaries. SkewTune is not designed to mitigate skew in the case where single invocations of the user-defined `map()` or `reduce()` functions take an extremely long time. To handle such cases, SkewTune would need to be extended with techniques such as those in the SkewReduce [81] system.

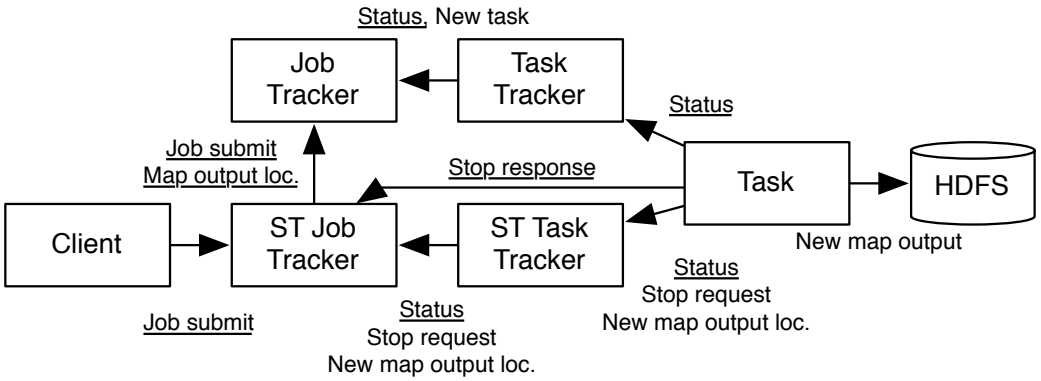


Figure 5.3: SkewTune Architecture. Each arrow is from sender to receiver. Messages related to mitigation are shown. Requests are underlined. Mitigator jobs are created and submitted to the job tracker by the SkewTune job tracker. Status is the progress report.

5.3 SkewTune for Hadoop

Overview: We implemented SkewTune on top of Hadoop 0.21.1. We modified core Hadoop classes related to (1) the child process, which runs the user supplied MapReduce application and (2) the Shuffle phase, which also runs in the child process. The only class we modified that runs in the Hadoop trackers is the `JobInProgress` class, which holds all information associated with a job. We added fields to track dependent jobs (*i.e.*, mitigator jobs) such that the map output is only cleaned up when there is no dependent job running.

The prototype consists of a job tracker and a task tracker analogous to those used in Hadoop. The child processes running with SkewTune report to both Hadoop and SkewTune trackers as shown in Figure 5.3. The SkewTune job tracker serves as the coordinator and is responsible for detecting and mitigating skew in the jobs submitted through its interface. The SkewTune task tracker serves as a middle tier that aggregates and delivers messages between the SkewTune job tracker and the Hadoop MapReduce tasks. When mitigating skew, the SkewTune job tracker executes a separate MapReduce job for each parallel data scan and for each mitigation.

Stopping a Straggler Task: When a straggler task has been chosen, SkewTune tries

to stop it by flagging a field in the heartbeat response message from the SkewReduce job tracker to its task tracker. Upon receiving the stop request, the task immediately reports back the currently processed record (*e.g.*, current offset in the input file or the current reduce key). If only a small amount of data remains to be processed, the task also runs the local data scan and returns the summary intervals. The stopped task then completes processing the current record and terminates. If a task is in the map-side sort, shuffle, or reduce-side sort phases, stopping the task is difficult because the outputs from disk spills, different tasks are intermingled at any point of those stages and it is hard to precisely define what are the remaining work in those stages in a compact manner. Tasks re-partitioned during those phases reports there is nothing to repartition and the coordinator makes decision as described in Section 5.2.2.

The current prototype only supports file-based inputs. Also, the `RecordReader` must implement an interface, `StoppableRecordReader`, to support stopping the process. When stop is requested, the record reader must return the current position in the record stream, and the remaining bytes if possible. If the stop was successful, the record reader immediately returns an end of stream on the next record request so that the map can start the SORT phase.

Repartitioning a Map Task: When SkewTune decides to repartition a map task, the map task runs the local scan (because map tasks are typically assigned with small amounts of data. It is possible to use the parallel scan if the size of remaining data is large and the input is replicated) and reports the summary intervals to the coordinator. The mitigators for a map task execute as map tasks within a new MapReduce job. They have the same map and, optionally combiner, functions.

We modify the original Map task implementation to sort and write the map output to HDFS when the task is a mitigator. Without this change, a map without reduce would skip the SORT phase. The map output index, *i.e.*, the information that reports which portion of the file is designated to which reduce task, is also written to HDFS for fault tolerance and sent to the SkewTune job tracker via a heartbeat message. The job tracker broadcasts the information about the mitigated map output to all reducers in the job.

Repartitioning a Reduce Task: To repartition a reduce task, the parallel scan job

(if it exists) and the mitigator job read map outputs from the Hadoop task tracker[2]. Thus, we implemented `InputSplit`, `TaskTrackerInputFormat` and `MapOutputRecordReader` to directly fetch the map output from task trackers. Our implementation uses the HDFS API to read the mitigated map outputs. `MapOutputRecordReader` skips over the previously processed reduce keys to ensure that only unprocessed data is scanned and repartitioned. For both jobs, we create one map task per node, per storage type (*i.e.*, task tracker and HDFS) so that each map task reads local data if the schedule permits it.

The map task in the mitigator job runs an identity function since all the data has already been processed. The partition function is replaced with a range partitioner provided by the SkewTune framework. The bucket information generated by the planner is compressed and encoded in the job specification. If a combiner exists in the original job, the map task also runs the same combiner to reduce the amount of data. Since the map is running the identity function, SkewTune knows that it can use more memory for the combiner and sort. Thus, it adjusts the corresponding configuration values appropriately. The reduce task runs unchanged.

Merging Mitigated Output: A mitigation job creates its output directory under the original output directory so that the mitigated output can be merged or disposed with the original output. The name of directory consists of the name of original task output and a suffix that identifies the mitigated job. The output can be merged in two ways. First, the following MapReduce job can read the output using an extended `CombinedFileInputFormat` which logically concatenates the mitigated output files with `CombinedInputSplit`. Or, on completion of the job, concatenates the files using HDFS concatenate operation [3].

Reducing Launch Overhead: Launching a new MapReduce job is relatively expensive compared to launching a job in a database management system (DBMS) because every task and job starts from scratch [108]. This overhead is critical for SkewTune since eliminating such overheads enables SkewTune to be used with relatively short jobs. An optimization we made to reduce the startup overhead of mitigator jobs is that SkewTune does not copy the

[2]Map output is served via HTTP by an embedded web server in the task tracker

[3]Implemented by HDFS-222 patch but requires that all intermediate blocks be full.

binaries and data per launch. Instead, SkewTune simply reuses the existing binaries and data copied to HDFS for the original job. This way, SkewTune can avoid the overhead of copying redundant files when launching a new parallel scan job as well as a new mitigator job.

Progress Monitoring and Estimation: The prototype implements Parallax to estimate time remaining [98]. We extend Parallax to handle multiple spills at the end of the Map phase. The extension is a simple analytical model of sort and spill as proposed in Li [86]. The estimate is calculated in the heartbeat thread of the child process and transmitted to the SkewTune task tracker with every report, roughly every 3 seconds. The SkewTune task tracker collects all reports from all local tasks, and submits them to SkewTune job tracker via a heartbeat message. Thus, there is an end-to-end delay from the map/reduce process to the job tracker that is double of the heartbeat interval.

Fault-tolerance: Fault-tolerance of SkewTune is mostly identical to that of Hadoop. The worst failure scenario in SkewTune is the same as in Hadoop: losing a map output due to a node failure. In this case, the lost map output has to be recomputed if there exists any reduce tasks that have not read it yet. Any map task failure or reduce task failure could be handled as if the failed tasks were experiencing skew. SkewTune may parallelize the re-execution but the current prototype does not implement this.

The failure of the coordinator could be handled similarly as in Hadoop except the coordinator has to persist repartitioning decisions so that the coordinator can make consistent decisions after recovery.

5.4 Evaluation

We evaluate the benefits of SkewTune when skew arises, SkewTune's robustness to initial job configuration parameters, and SkewTune's overhead in the absence of skew. We find that SkewTune delivers up to a factor of 4X improvement on real datasets and real UDOs. It also significantly reduces runtime variability. Further, the overhead of SkewTune in the absence of skew is shown to be minimal.

All experiments are performed on a twenty-node cluster running Hadoop 0.21.1 with a separate master node. Each node uses two 2 GHz quad-core CPUs, 16 GB of RAM, and

two 750 GB SATA disk drives. All nodes are used as both compute and storage nodes. The HDFS block size is set to 128 MB and each node is configured to run at most four map tasks and four reduce tasks concurrently.

We evaluate SkewTune using the following applications.

Inverted Index (II): An inverted index is a popular data structure used for Web search. We implemented a MapReduce job that builds an inverted index from the full English Wikipedia archive and generates a compressed bit vector for each word. The Potter word stemming algorithm is used to post-process the text during the map phase [4]. The RADIX partitioner is used to map letters of the alphabet to reducers and to produce a lexicographically ordered output. The total data size is 13 GB.

PageRank (PR): PageRank [25] is a popular link analysis algorithm that assigns weights (ranks) to each vertex in a graph by iteratively aggregating the weights of its inbound neighbors. We take the PageRank implementation from Cloud 9 [72] and apply it to the freebase dataset [55]. The total input data size is 2.1 GB.

CloudBurst (CB): CloudBurst [116] is a MapReduce implementation of the RMAP algorithm for short-read gene alignment[5]. CloudBurst aligns a set of genome sequence reads with a reference sequence. We take the CloudBurst application and use it to process a methylotroph dataset [73]. The total input data size is 1.1 GB.

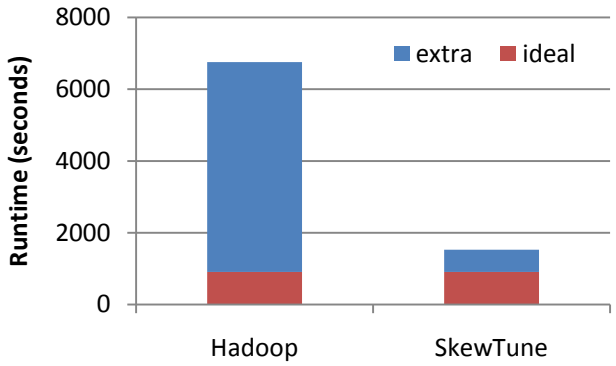
5.4.1 Skew Mitigation Performance

The first question that we ask is how well SkewTune mitigates skew.

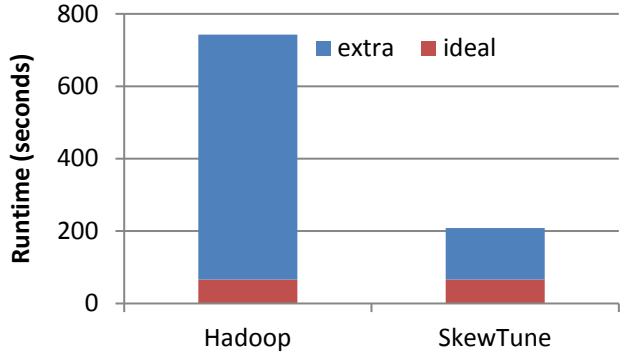
Figure 5.4(a) shows the runtime for the reduce phase of the Inverted Index application. When using vanilla Hadoop, the reduce phase runs across 27 reducers (one per letter of the alphabet and one for special characters) and completes in 1 hour and 52 minutes. With SkewTune, as soon as the reduce phase starts, SkewTune notices that resources are available (there are a total of 80 reduce slots). It thus partitions the 27 tasks across the available

[4]We use a bit vector implementation and a stemming algorithm from the Apache Lucene open source search engine.

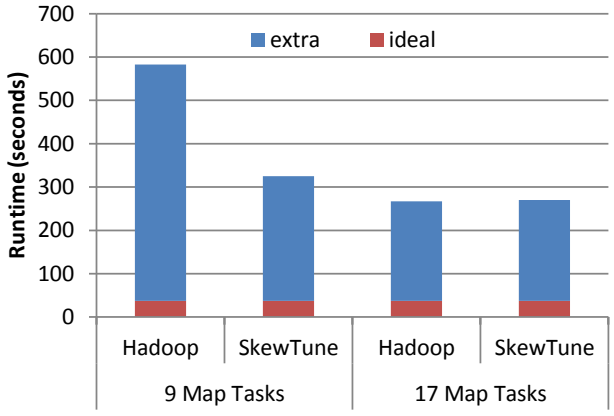
[5]`http://rulai.cshl.edu/rmap/`



(a) Reduce Phase of Inverted Index with RADIX partitioner



(b) Map Phase of CloudBurst



(c) Map Phase of PageRank

Figure 5.4: UDO runtime with and without SkewTune.

slots until the cluster becomes fully occupied. The runtime drops to only 25 minutes, a factor of 4.5 faster. This experiment demonstrates that, with SkewTune, a user can focus on the application logic when implementing her UDO. She does not need to worry about the cluster details (*e.g.*, how to write the application to use N reducers instead of the natural 27).

In the figure, we also show the ideal execution time for the job. This execution time is derived from the logs of the vanilla Hadoop execution: we compute the minimal runtime that could be achieved assuming zero overhead and a perfectly accurate cost model driving the load re-balancing decisions. In the figure, we see that SkewTune adds a significant overhead compared to this ideal execution time. The key reasons for the extra latency compared with ideal are scheduling overheads and an uneven load distribution due to inaccuracies in SkewTune's simple runtime estimator. SkewTune does, however, improve the total runtimes greatly compared with vanilla Hadoop. In the rest of this section, we always decompose the runtime into *ideal* time and *extra* time. The latter accounts for all real overheads of the system and possible resource under utilization.

Figure 5.4(b) shows the runtime for the map phase of CloudBurst. This application uses all map slots. Hence, the cluster starts off fully utilized. However, the mappers process two datasets: the sequence reads and the reference genome. All map tasks assigned to process the former complete in under a minute. With vanilla Hadoop, the job then waits for the mappers processing the reference dataset to complete. In contrast, SkewTune re-balances the load of the mappers processing the reference dataset, which improves the completion time from 12 minutes to 3 minutes (ideal time is 66 seconds). This application is a classical example of skew and it demonstrates SkewTune's ability to both detect and mitigate that skew. Notice that skew arises even though all mappers are initially assigned the same amount of data (in bytes).

Finally, we demonstrate SkewTune's ability to help users avoid the negative performance implications of mis-configuring their jobs. Figure 5.4(c) shows the runtime for the map phase of PageRank. The figure shows two configurations: a good configuration and a worst-case configuration. In the good case, vanilla Hadoop and SkewTune perform similarly. However, if the job is mis-configured, vanilla Hadoop leads to a significantly longer completion

time while SkewTune maintains a consistent performance. To create the bad configuration, we simply changed the input data order: we sorted the nodes in the graph by increasing order of outdegree. While in practice a user may not necessarily hit the worst-case configuration for this application, the experiment shows that vanilla Hadoop is sensitive to user mis-configurations, unlucky data orders, and other unfortunate conditions. In contrast, SkewTune delivers high performance systematically, independent of these initial conditions.

5.4.2 Performance Consistency

In this section, we further study the consistency of the performance that SkewTune delivers. For this, we run the CloudBurst and PageRank applications but we vary the initial number of tasks. Figure 5.5 shows the results for the map phase of CloudBurst using either 80 or 801 mappers and PageRank using either 9 or 17 mappers. As the figure shows, Vanilla Hadoop is sensitive to these configuration parameters with up to a 7X difference in runtimes. In contrast, SkewTune's performance is significantly more stable with performance differences within 50%. The figure shows, however, that for configurations without skew in PageRank, SkewTune yields a runtime higher than that of vanilla Hadoop (3 s more). This is due to inaccurate time-remaining estimates: SkewTune missed the timing to mitigate skew of the longest map task and made an unnecessary split of another task. The overhead, however, is negligible.

5.4.3 Skew Mitigation Overhead

To measure some of SkewTune's overheads, we re-run the same applications as above, but we tune them to ensure low runtimes with vanilla Hadoop. We make the following tunings. For CloudBurst, we configure the number of map and reduce tasks exactly as the author recommends: We use 10 times as many map tasks and 2 times as many reduce tasks as slots. In the experiment, we thus get 801 map tasks (the last task is assigned only a small amount of data due to rounding in size) and 160 reduce tasks. For the Inverted Index, we use a hash partitioner and spread the reduce input across 140 tasks. Finally, for PageRank, we use 17 map and 17 reduce tasks with 128 MB chunks. This configuration differs from

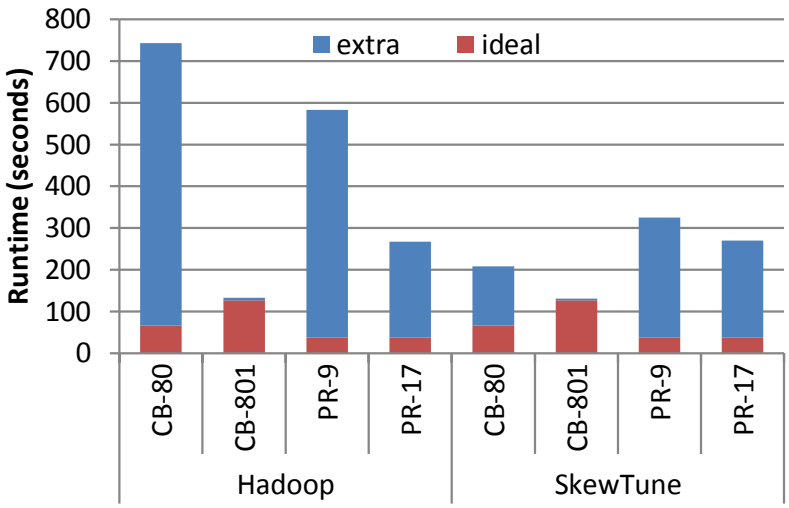


Figure 5.5: Performance Consistency of Map Phase: For both PageRank (PR) and CloudBurst (CB), SkewTune delivers high-performance consistently, while Hadoop is sensitive to the initial configuration (here, the number of map tasks).

the worst-case configuration in the ordering of data (the original ordering of the dataset vs sorted by record size) and a smaller chunk size (128MB vs. 256MB).

Figures 5.6 and 5.7 show the results. As the figures show, SkewTune adds overhead but that overhead is small. In most cases when applications are already well-tuned and do not exhibit skew, the slots remain busy. SkewTune has few opportunities to improve performance or incur repartitioning overhead. As a result, performance may improve only slightly as in the case of the CloudBurst and Inverted Index reduce phases. In other cases, the runtime can slightly increase. Also with shorter overall runtimes, the overheads of stopping, planning, and re-partitioning become more pronounced. Errors in progress estimation also have more visible effects as does any unnecessarily re-partitioning of nearly completed tasks.

In Figure 5.7, we also show the result of the REHASH technique, where we replace SkewTune's range partitioning with hash partitioning thus avoiding the need to scan the

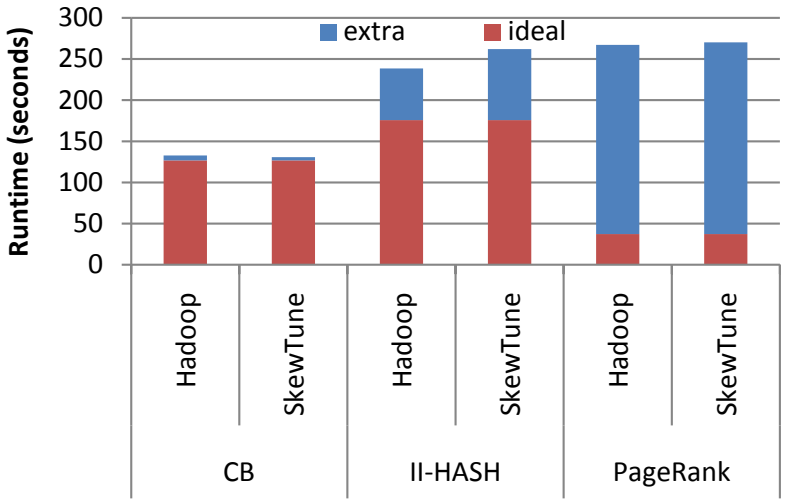


Figure 5.6: Runtime of Map Phase without Skew

remaining input data. Overall, REHASH performs slightly better than SkewTune due to its reduced overhead but it requires an extra job to recover the ordering (note that the numbers do not include such extra jobs!). SkewTune is only marginally slower than REHASH but it preserves the output order.

Detailed Mitigation Overhead Analysis: We further analyze the overhead of mitigating the skew of a single straggler by analyzing the execution logs of 32 map task mitigations and 64 reduce task mitigations from our three test applications. Overall, in these experiments, the current SkewTune prototype incurs approximately 15 sec overhead for map task skew mitigation and 30 sec for reduce tasks.

Table 5.2 shows the breakdown of the overhead. Interestingly, the mitigator planning phase takes less than 200 ms. It hardly incurs any overhead due to the compact summary information. We ran extra experiments (not shown due to space constraints), where we varied the interval granularity. We found the PLAN phase to be consistently fast and below 500 ms in all configurations. The most significant overhead component is the data scan, which takes approximately 10 to 15 sec for a local scan. This overhead grows linearly with the size of the input data. Because SkewTune repartitions more data for reduce tasks than

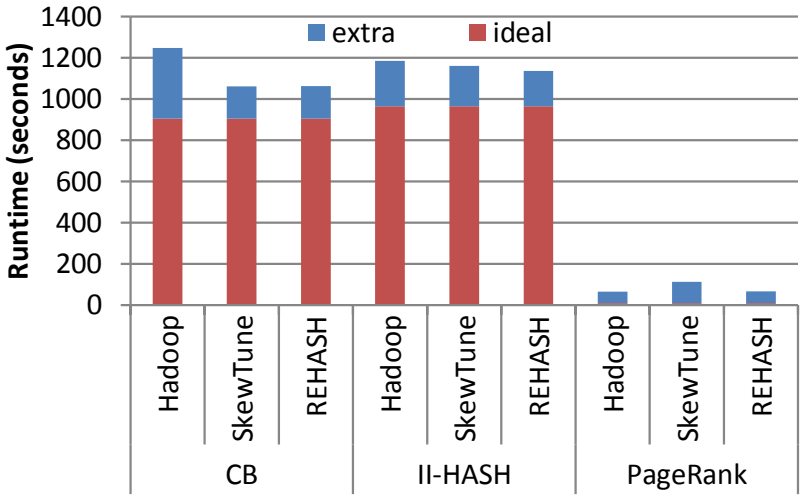


Figure 5.7: Runtime of Reduce Phase without Skew

map tasks in these experiments, it follows that the total overhead is larger for reduce tasks. With the same applications and datasets, parallel scans take between 20 and 22 sec. This includes the startup and tear down overhead of the MapReduce job as well as shuffling and sorting overheads when scanning map outputs. This overhead also grows linearly but with a much smaller slope as we discuss below.

"< Compute" represents the time between mitigator planning and the resumption of the data computation. In case of map mitigation, this time only includes the overhead of starting a new job. For reduce mitigation, the overhead includes another scan of the data to repartition and re-shuffle that data.

Overhead of Local Scan vs. Parallel Scan: In all three applications and datasets, the size of remaining data during skew mitigation is small (< 1 GB). Thus, SkewTune always performs a local scan rather than a parallel scan. To evaluate the trade-off between the two approaches, we compared the performance of the two scan strategies using a synthetic workload. Figure 5.8 shows the results. We generated random datasets with different sizes and evenly distributed them across all 20 nodes. To simulate a realistic environment, we loaded all the disks using two background writer processes per disk and dropped the disk

Type	Scan	Plan	< Compute	Input Bytes
Map	8.0s (3.0)	0.19s (0.08)	5.01s (3.83)	84MB (55)
Reduce	15s (15.0)	0.18s (0.19)	15.7s (10.4)	140MB (175)

Table 5.2: Mitigation Overhead Statistics. The average and standard deviation (number in parentheses) in seconds for each mitigation step. Size of re-partitioned data. "< Compute" represents time until the actual processing resumes. Scans are all local scans.

cache before the scan. The timing of parallel scan includes the MapReduce job startup and cleanup overhead. In our 20 node cluster, parallel scan performs better than local scan if the size of remaining data is greater than 1 GB. With smaller data, the MapReduce job overhead dominates the I/O time. However, once the data becomes large enough, the overhead pays off by reading a small amount of data per disk while local scan has to sequentially read the data from a single disk. Clearly, the gain will diminish if there exists a significant skew in the amount of distributed input data. For example, for 8 GB of data, local scan takes 890 s but parallel scan takes 679 s when a node has 7.2 GB of data.

Summary: The above experiments show that SkewTune effectively mitigates skew whether it is intrinsic to the application, caused by a misconfiguration, or due to an unfortunate input data order. SkewTune also delivers consistently fast runtimes independent of initial job configuration parameters. SkewTune's overhead is small to none when there is no skew. Finally, the greatest overhead component of re-partitioning a straggler's data comes from the data scans necessary for planning and re-allocating the data. SkewTune's ability to perform these scans in parallel when possible, however, effectively keeps these overheads low even when large datasets need to be repartitioned.

5.5 Conclusion

In this chapter, we presented SkewTune, a system that automatically mitigates skew in a broad class of user defined operations implemented as MapReduce jobs. SkewTune requires no input from users. It is broadly applicable as it makes no assumptions about the cause of

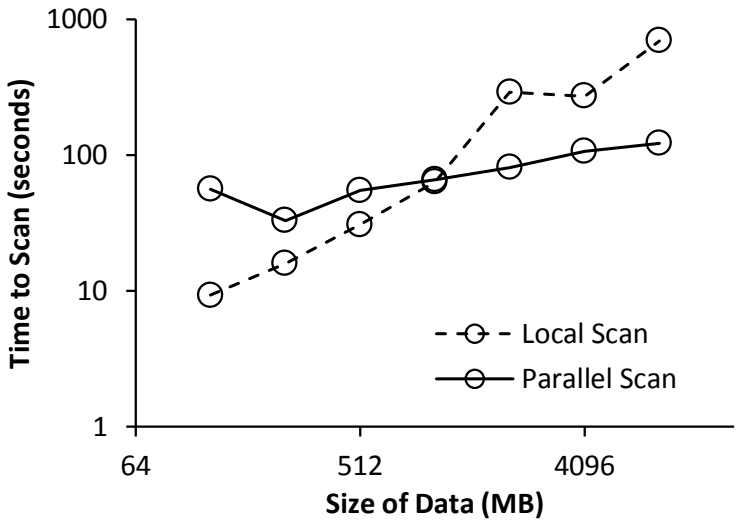


Figure 5.8: Overhead of Local Scan vs. Parallel Scan. Time was measured under heavy disk load. For small data sizes (< 1 GB), local scan is faster. For large data sizes (> 1 GB), parallel scan is faster.

the skew but instead observes the job execution and re-balances load as resources become available. SkewTune is also capable of preserving the order and partitioning properties of the output of the original unoptimized job, making it transparently compatible with existing code, even in the context of complex workflows and advanced MapReduce algorithms.

Experimental results show that SkewTune can deliver a factor of 4X improvement over Hadoop on real and representative datasets and real, non-trivial UDOs. At the same time, it adds little to no overhead when skew is not present. Finally, it provides for much more consistent job execution times for jobs that sometimes incur skew thereby enabling more predictable performance.

Chapter 6

RELATED WORK

In any parallel system, improper skew handling can counter all the benefits of parallel processing, so skew must be handled effectively.

6.1 Parallel UDO Evaluation Systems

There are many systems that can run UDOs in parallel.

Parallel DBMSs have been supporting UDOs for a long time [12, 58, 101, 124, 128, 137]. Parallel DBMSs typically support two types of UDOs: user-defined functions (UDFs) and stored procedures. UDFs can be used as expressions in a SQL query. There are three types of UDFs: scalar functions (*i.e.*, returning a scalar value), table functions (*i.e.*, returning a relational table), and aggregate functions (*i.e.*, processing a group of tuples and returning a scalar value). A stored procedure is a procedure executed entirely in the database engine. In parallel DBMSs, both UDOs and statements in a stored procedure are executed in parallel. Although parallel DBMSs do support UDOs, the primary focus of parallel DBMSs is running relational queries faster.

The MapReduce system has proposed a programming model of UDOs and an execution strategy of UDOs in a cluster of commodity hardware [37]. The MapReduce system has inspired many other systems that improve MapReduce in different ways including more general job descriptions (*i.e.*, a directed acyclic graph of operators) and adapting the ideas of parallel DBMSs [14, 22, 43, 61, 68, 115]. Just like the SQL in parallel DBMSs, many high-level declarative interfaces on top of MapReduce-like systems have been proposed to improve productivity of writing parallel data analysis programs [18, 27, 104, 109, 131, 147]. All those systems and declarative interfaces support UDOs by default.

Several parallel DBMSs support MapReduce interface in addition to the traditional SQL interface [12, 48, 58, 137].

6.2 Skew Handling through Skew-Resilient Implementation

As we showed in Chapter 2, a skew-resilient implementation is possible if the algorithm and the system are well understood. In this section, we review skew-resilient implementations designed for relational operators (join and aggregate) and parallel scientific simulations.

6.2.1 Join

In parallel database research, the skew problem has been extensively researched by many research groups, especially in the context of the join operator.

Walton *et al.* studied taxonomy of data skew in a parallel join [138]. The taxonomy of Walton *et al.* inspired the taxonomy of skew in aggregation [117] and taxonomy of skew in MapReduce as presented in Chapter 3.

Many different parallel join algorithms to address skew have been proposed.

Kitsuregawa *et al.* proposed a bucket spreading hash join policy to handle data skew [77]. During the partitioning phase, a specially designed network switch evenly distributes the fragments of a hash bucket among all processors. During the join phase, each bucket is dynamically assigned to a processor. The processor first collects all distributed fragments of the assigned bucket, and then performs the join algorithm. The concurrent bucket collection is coordinated to prevent I/O contention.

Hua *et al.* proposed three adaptive join algorithms with partition tuning [65, 66]. The goal of partition tuning is assigning buckets to processors so that all processors have roughly equal amounts of data (*i.e.*, total size of buckets assigned to each processor). The tuple interleaving parallel hash join is a variant of a bucket spread join that does not require a special network switch. The adaptive load balancing parallel hash join is a conventional hash join with partition tuning between the split and join phases. The extended adaptive load balancing parallel hash join is a variant of adaptive load balancing parallel hash join that defers tuple transfer after partition tuning.

Wolf *et al.* proposed a scheduling-based algorithms for parallel joins [142, 143]. The key idea is that estimating the cost of join partitions based on statistics then applying the longest processing time (LPT) scheduling algorithm to determine processing order among

join partitions.

DeWitt *et al.* proposed a practical approach to handle skew in a parallel join [42]. They considered combinations of range partitioning (with subset-replication, weighting), load scheduling (*e.g.*, round-robin, LPT), and virtual processors (*i.e.*, create many logical partitions and schedule multiple partitions per physical processor). For range partitioning, DeWitt *et al.* proposed an efficient sampling technique to estimate the distribution of the join key value. For LPT scheduling, they used a simple cost model to estimate the cost of each partition. This technique can handle redistribution skew but not the join product skew.

Shatdal *et al.* proposed a join algorithm that handles join product skew with distributed shared memory [118]. During the join phase, an idle processor that has just completed its assigned join work chooses a busy processor and steals the remaining join work. The algorithm requires shared states (*i.e.*, split table per processor [42]) and shared data structures (*i.e.*, hash table) among all the processors. The distributed shared memory abstraction thus greatly simplifies the implementation.

Li *et al.* proposed block-based sort merge join algorithms that impose little overhead when there is no skew and perform efficiently with heavily skewed workload [87].

Xu *et al.* proposed a parallel join approach, partial redistribution partial duplication (PRPD) to handle data skew [145]. The PRPD approach first splits the rows of joining relation into three disjoint groups that are handled differently (redistributed, duplicated, or kept local). The rows having skewed values are kept local, the rows bearing skewed values of joining attributes in the other relation are duplicated, and the remaining rows are redistributed as in an ordinary parallel hash join. The final join is computed by union of three joins (two replicated-joins for skewed join values of each relation, and an ordinary parallel join for non-skewed values). Xu *et al.* also proposed a parallel outer join algorithm in the presence of data skew [144].

In MapReduce, a join is just another UDO, so there are many different ways to implement a join. Pig, a declarative layer of Hadoop, implements an algorithm proposed in parallel database literature [42] to handle data skew in a join algorithm [52]. Blanas *et al.* surveyed and compared different join implementations in MapReduce [20]. Afrati and Ullman studied

the multi-way join problem in MapReduce and proposed an algorithm that minimizes total communication cost [3]. Koutris *et al.* studied the complexity of evaluating conjunctive query (*i.e.*, multi-way join) in a MapReduce-like model [80]. Okcan *et al.* studied a theta-join in MapReduce [102]. Lin *et al.* proposed MapReduce join implementations using column-oriented storage [91].

Users can easily implement more advanced joins in MapReduce.

CloudBurst is a bioinformatics application that aligns DNA sequence reads with a reference database [116]. CloudBurst is implemented in Hadoop and performs a similarity join between a DNA sequence reads dataset and the reference database. CloudBurst balances load using domain knowledge; frequently occurring, low-complexity sequences (calculated using domain knowledge) are distributed among multiple reduce tasks.

Vernica *et al.* and Metwally *et al.* studied set-similarity joins in MapReduce [96, 136]. Vernica *et al.* studied the set-similarity join based on prefix filtering with several MapReduce-specific optimizations [136]. To handle skew, Vernica *et al.* first scanned the input data to collect statistics on frequencies and load balance according to the statistics. Metwally *et al.* proposed V-SMART-Join framework for all-pair similarity joins of sets, multisets, and vectors [96]. Metwally *et al.* carefully analyzed set similarity measures and extracted a common structure in computing similarities, and then designed a series of MapReduce steps according to the structure. Metwally *et al.* also analyzed skew that may occur in each step, and described skew handling strategies.

Spatial joins are also implemented in MapReduce [149, 151]. SJMR is a general spatial join framework in MapReduce [151]. Zhang *et al.* proposed a k-nearest neighbor join implemented in MapReduce [149]. To handle skew, both spatial joins partition the spatial input data using space-filling curves.

6.2.2 Aggregate

Shatdal *et al.* investigated skew problems in the parallel aggregation [117]. Shatdal *et al.* identified two skew problems in parallel aggregation: input skew and output skew. Input skew problems arise when the number of groups per node is equal but the number of tuples

per node is different (*i.e.*, some nodes process more input data than other nodes). The output skew arises when the number of tuples per node is equal but the number of groups per node is different (*i.e.*, some nodes produce more output than other nodes). Two adaptive algorithms were proposed. The first algorithm (adaptive two phase) initially assumes that there exists a small number of aggregation groups and runs two-phase aggregation (*i.e.*, local aggregation followed by a global aggregation). If there exist too many groups, the algorithm is switched to a repartitioning algorithm (*i.e.*, redistribute input table based on group attribute value), and the global aggregation processes both intermediate aggregation results or raw data tuples. The second algorithm (adaptive repartition) starts with a different assumption from that of the adaptive two phase algorithm. The algorithm first assumes that there exists a large number of groups and thus starts with the repartitioning algorithm. If a node detects too small a number of groups, the aggregation is switched to the adaptive two phase algorithm. Both algorithms are applicable to distributive or algebraic aggregate operators which are relatively inexpensive and easy to migrate into the intermediate state [57].

Yu *et al.* surveyed and evaluated different aggregation APIs and execution strategies in distributed systems [146]. Rusu *et al.* proposed the GLADE system, a distributed aggregation framework based on a generalized linear aggregate (GLA) [115]. Neither Yu *et al.* nor Rusu *et al.* considered data skew in evaluation. Li *et al.* proposed a hash-based incremental analysis platform and implemented a prototype using Hadoop [86].

6.2.3 Parallel Scientific Simulation

Scientific simulation communities have long studied load imbalance problems in parallel systems. Just as in parallel database systems, there exist many mature infrastructures to run parallel simulations in a scalable manner [39, 103, 107]. The primary technique for attacking skew is adaptively repartitioning (or regridding) the data by periodically monitoring the application runtime statistics or through explicit error measures of the parallel simulation [63]. Several cosmological simulations partition the workload based on gravitational potential and construct a tree to balance parallel spatial index lookup and computation [125].

Summary

In this section, we reviewed many skew-resilient implementations of important algorithms in database and scientific simulation. All such implementations are highly optimized based on deep understandings of target applications. As shown in Chapter 2, developing a skew-resilient implementation requires significant efforts. The two proposed approaches SkewReduce and SkewTune support a broad class of applications (*e.g.*, feature extracting applications for SkewReduce and MapReduce applications for SkewTune). By carefully exploiting the common structure of applications and the properties of the execution engine, SkewReduce and SkewTune can execute skew-prone UDOs in a scalable manner without developing a skew-resilient implementation every time.

6.3 Skew Handling for UDOs

The most closely related work to this thesis regards skew-handling in MapReduce.

6.3.1 Skew in MapReduce

MapReduce and similar large scale data analysis platforms handle machine skew using speculative execution [37, 61, 68], which simply re-executes slow tasks on multiple machines and takes the result of the first replica to complete. Speculative execution is effective in heterogeneous computing environments or when machines fail. However, it is not effective against data skew because rerunning the skewed data partition even on a faster machine can still yield a very high response time. Lin analyzed such impact of data skew of a MapReduce program [89].

Qiu *et al.* implemented three applications for bioinformatics using cloud technologies and reported their experience and measurement results [111]. Although they also found skew problems in two applications, they discussed a potential solution rather than tackling the problem.

For data and computation imbalance problems, Ke *et al.* also found these problems are prevalent in a variety of industrial applications [76]. Ananthanarayanan *et al.* studied the causes to outlier (*i.e.*, straggler) tasks and ways of mitigating outlier tasks caused by data

imbalances or resource contentions [7].

6.3.2 Adaptive MapReduce

There are several proposals to handle skew by adaptively executing UDOs. These proposals are closely related to SkewTune.

Ibrahim *et al.* and Gufler *et al.* studied data skew in the reduce phase [59, 67]. Both approaches schedule reduce keys to the reduce tasks based on cost models. Also, the reduce key scheduling does not preserve the order as in the original reduce output. SkewTune not only addresses skew in both the map and reduce phases but also minimizes the side-effect of skew mitigation by preserving input order.

In the follow-up work, Gufler *et al.* proposed the TopCluster approach to construct a histogram of all reduce keys to identify skewed reduce keys [60]. The TopCluster approach is similar to reconciling the result of the parallel scan in SkewTune. TopCluster eagerly monitors, detects, and mitigates reduce skew, while SkewTune lazily detects and mitigates skew. Also, in SkewTune, the planning is done using exact information, if possible.

Vernica *et al.* proposed an adaptive MapReduce system using situation-aware mappers that continuously monitor the execution of mappers and adaptively resplit the map input data [135]. Also, with an adaptive combiner and partitioner, the system also tries to balance the reduce input. However, the situation-aware mappers can not handle computational skew at the reducers, where some key-groups take longer to process than others.

Rao *et al.* proposed the Sailfish system, a drop-in replacement of Hadoop [112]. Sailfish uses $\mathcal{I}$-files, aggregated intermediate data stored in a distributed file system, to manage intermediate data between Map and Reduce phases. With $\mathcal{I}$-files, Sailfish not only improves performance, but also enables users to choose the number of reduce tasks dynamically.

There are three key differences between the above proposals and SkewTune. First, the above approaches *eagerly* monitor the execution and data to detect and react to skew. On the other hand, SkewTune *lazily* detect and react to skew. The second difference is transparency. SkewTune is more transparent than the other approaches by strictly preserving the input data order of mitigation tasks. The other approaches may require an extra job

to reconstruct the original output. Lastly, SkewTune can handle different types of skew in both map and reduce phases while the others are specialized to address specific types of skew in a specific phase. SkewTune is complementary to the other approaches. SkewTune can handle anything left by each of the above systems in a transparent manner.

Agarwal *et al.* proposed RoPE, an adaptive query processing (AQP) technique for SCOPE [27], including UDOs [4]. Previous AQP research in databases mostly focused on executing relational queries by correcting optimizer mistakes and dealing with unknown statistics [13, 38]. RoPE can adaptively increase or decrease the degree of parallelism of current and descendant operators, but the changes are effective to only the unexecuted part of the plan. SkewTune changes a running UDO partition and is complementary to RoPE.

Chapter 7

CONCLUSION AND FUTURE WORK

The emerging big-data analysis trend requires complex data processing at large scale. The user-defined operations (UDOs) are a powerful mechanism to express such complex processing, and the existing parallel processing engines (*e.g.*, parallel DBMSs and MapReduce-like systems) can execute UDOs at large scale in a cluster of commodity hardware. In this thesis, we tackle one of the challenges in parallel UDO evaluation: skew.

Problem

Skew refers to a significant variance between different partitions of an operator executing in parallel. With greater skew, the benefit of parallelization quickly diminishes because the job completion time is dominated by the slowest task, which may run orders of magnitude longer than its peer tasks. With UDOs, handling skew is more challenging than with relational operators because a UDO is a black box to the execution engine, and thus the user is fully responsible for writing skew-resistant code. While skew can arise in the execution of a UDO both in a parallel DBMS and in MapReduce, in this dissertation, we focused on the skew problem in MapReduce-type systems because these systems have been designed specifically for UDO implementation and are known to be easier to use for this purpose [108].

Summary of Contributions

In this thesis, we first studied the problem of skew in UDOs through a detailed case study on porting an existing data analysis algorithm to Dryad, a MapReduce-like execution engine that provides a more flexible API than MapReduce. We showed that there may exist a skew-resistant implementation of an algorithm, but the design and development of such implementation require significant efforts. Furthermore, through a measurement study of three Hadoop cluster workloads, we found that more than 40% of jobs running longer than

five minutes had experienced skew problems. The majority of slow tasks ran twice as long as their median peer tasks, and there were tasks that ran orders of magnitude longer than the median runtime. Also, we found that the de facto countermeasure (*i.e.*, speculative execution) was effective in improving the completion time in only 20% of instances.

We proposed two techniques to manage skew in parallel UDO evaluation: *SkewReduce* and *SkewTune*.

SkewReduce is a skew-avoidance technique that statically optimizes data partitioning for a specific class of applications, feature extracting applications. The key insight of SkewReduce is that users can reason about the complexity of the computations even though they are not experts in parallel programing. Based on this intuition, the SkewReduce partition optimizer uses user-defined cost models (black-box functions that estimate the computation complexity of a given data) to identify expensive parts of the data and appropriately increase the number of partitions around these parts.Thus, the optimizer can reduce the impact of skew using more finer-grained data partitions when processing data susceptible to skew. The optimization is greedy-top-down. At each iteration, the optimizer uses the user-defined cost models to find the best hyperplane that divides the data into two subpartitions of approximately equal costs. After each split, the optimizer evaluates the new expected runtime by simulating the UDO execution with the new data partition. If the runtime improves, the optimizer accepts the split. The optimizer continues to work on the most expensive partition until no further split improves the runtime. SkewReduce can improve job completion times of feature extracting applications by a factor of up to 8x compared with the default data partitioning strategy of MapReduce.

SkewTune is a dynamic skew mitigation technique for data-parallel UDOs, that process input records independently: *i.e.*, MapReduce jobs where both the map and reduce functions do not keep state between consecutive invocations. SkewTune first monitors the execution of a UDO and estimates the remaining time of each task. Whenever a compute node becomes idle, SkewTune selects the task with the longest remaining time and checks whether re-partitioning the remaining input data of the task improves the overall completion time of the job. SkewTune then repartitions the remaining input data of the selected task in a way that fully utilizes the cluster based on the time remaining estimates

of all concurrent tasks. SkewTune repeats the monitoring-mitigation cycle until there are no remaining tasks. The greatest benefit of SkewTune is its transparency. Users do not need to modify existing MapReduce applications for SkewTune. Also, SkewTune does not require cost models. Instead, SkewTune collects the information necessary for mitigation by monitoring the execution of a job. Finally, the carefully designed input data repartitioning technique guarantees that the original output without skew mitigation is trivially reconstructed by concatenating the output of the tasks of the modified job and is thus transparent to downstream applications that consume that output. In the evaluation of real MapReduce applications, SkewTune improved a job completion time by a factor of 4 without any code-level modification. SkewReduce and SkewTune together can provide a practical framework for parallel UDO evaluation especially in the cloud. First, users can plan the executions in a cloud service using the SkewReduce optimizer by testing different cluster sizes. At runtime, SkewTune can handle dynamic runtime events such as failures, interference, and estimation errors. Also, SkewTune can leverage opportunistic resources, such as spot instances of Amazon EC2 [6], if the users want to accelerate the job execution.

Overall, we found that skew in parallel UDO is a frequent yet challenging problem through a case study and a measurement study. The two proposed techniques, SkewReduce and SkewTune, demonstrate that the impact of skew in parallel UDOs can be reduced statically and dynamically at a greatly reduced user efforts.

7.1 Short-Term Future Work

The most straightforward future work is to generalize SkewReduce to MapReduce applications beyond feature extracting applications. For this, the SkewReduce API needs to be redefined for map and reduce. The partition optimization can be carried out along the key dimension (*e.g.*, the reduce key for the reduce and the offset in the input file for map). Second, this generalized SkewReduce could be integrated with SkewTune to handle skew in MapReduce applications. The generalized SkewReduce generates partition plans for both map tasks and reduce tasks before execution. At runtime, SkewTune can exploit the extra partition, merge, and finalize functions (*i.e.*, generalized SkewReduce API for MapReduce) to handle an expensive input skew better through parallelization. If a user provides cost

models, then the user-defined cost model can override the default progress estimator of SkewTune. If both user-defined cost models and extra API are given for the MapReduce application, the SkewReduce optimization can override the SkewTune repartition planning. An interesting extension of SkewReduce static optimization and SkewTune repartition planning is taking failures into account during the optimization (*e.g.*, given a priori node failure probability, what is a good partitioning plan?). Lastly, evaluating such a system in a large shared cluster of 1,000 or more machines would be an interesting study.

7.2 Long-Term Future Work

7.2.1 Skew Handling in Other Parallel Systems

The thesis focuses on skew handling in a shared-nothing cluster, but there are many emerging technologies for parallel processing such as multicore CPUs, GPGPUs, distributed shared-memory enabled by the fast network interconnects (*e.g.*, InfiniBand and 10Gbps Ethernet), and advanced shared-memory systems (*e.g.*, Cray XMT).

Both SkewReduce and SkewTune demonstrate that revising the degree of parallelism (either statically or dynamically) is effective in handling skew. This is true for any parallel system. SkewReduce works regardless of the architecture of the execution engine as the technique is static planning before execution, but it requires tuning parameters such as task scheduling delay, which may vary per architecture.

SkewTune works in other systems, but the implementation may vary to leverage the properties of the underlying architectures fully. For example, in a shared-nothing architecture, scan and repartition during skew mitigation are expensive as both involve disk and network I/O. In a shared-memory system, however, the overhead of scan and repartition are significantly lower than in a shared-nothing system. Sharing state and coordinating tasks (*i.e.*, threads) are also easier and faster in a shared-memory system than a shared-nothing system thus the overall implementation could be simpler. However, different problems and optimizations (*e.g.*, data layout in memory, cache coherency, and cache miss) may be more important in other parallel systems than in a shared-nothing system. SkewTune is designed for disk-oriented MapReduce execution, and so does sequential I/O as much as possible

(*e.g.*, the MapReduce execution strategy is optimized for sequential I/O. Both parallel and local scan in SkewTune sequentially access the data). Fortunately, such sequential property of SkewTune is also preferred in other architectures.

It would be interesting to implement SkewTune's approach in other architectures and compare the trade-off and optimizations across different architectures. It would also be interesting to apply the ideas to other types of parallel processing engines (*e.g.*, parallel DBMSs and graph processing engines [9, 93, 95]) and study trade-offs and optimizations specific to different programming models.

7.2.2 Skew-Tolerant System Design

Skew can arise in any parallel system. If any of the assumptions that a parallel system makes do not hold, skew is likely to arise. Some common assumptions include uniform data "value" distribution, constant processing time per input byte/record, over-reliance on default values/strategies, and the failure model. The system designer should carefully evaluate such assumptions and check whether each assumption is the norm rather than an exception. To do so, a thorough understanding of workloads is important.

If skew is the norm in a system and for a given workload, as shown in Chapter 3 for MapReduce, the system designer should include a proper countermeasure so that either the system or the user can easily tolerate the skew. As shown in the thesis, flexible partitioning and scheduling of parallel tasks are promising countermeasures to skew and so should be considered in the design when the skew is anticipated in the workload especially with user-defined operations.

Identifying the root cause of skew is also important in a skew-tolerant system because the proper way to handle skew depends on the source of the problem. For example, if a failing disk is the problem, then it is the best to avoid using the disk. The scheduler should be aware of this and schedule mitigation tasks in other nodes which preferably have a replica of the input data. To isolate the root cause, the system needs to monitor various parts of the system including application performance counters and make a scheduling decision based on the holistic observations. Clearly, this approach complicates scheduling decisions

and increases the monitoring overhead. Thus, using a combination of scalable monitoring, detection, and scheduling algorithms based on holistic information is an interesting research direction.

7.2.3 Oblivious Skew Handling

Ideally, users should be completely oblivious to skew handling. Although SkewReduce and SkewTune somewhat disburden users with writing skew-resistant code, there still exists much room to improve by incorporating recent advances in programming languages and compilers.

Both SkewReduce and SkewTune rely on cost models. In SkewReduce, the system expects that the models are specified by the users. In SkewTune, the system uses a general model based on input byte consumption rate. As shown in Figure 4.4, a high-fidelity cost model can yield a better plan than a low-fidelity model. An interesting future work would be to construct a progress estimator and cost model through dynamic instrumentation and code analysis at runtime. As shown in Ren *et al.* [114], many UDOs are written in high-level programming languages such as Java, C#, and Python in which dynamic instrumentation and code analysis are much easier than in the native programming languages such as C/C++. Also, extracting relevant UDO features (*e.g.*, blocking vs. non-blocking, pure function, output ordering) for optimization through analysis would be interesting. There exists several early works along this direction [30, 62, 70, 150].

7.3 Final Remarks

Skew is a challenging problem in a parallel system. As shown in the thesis, it often requires significant efforts to address skew in a parallel application. This dissertation contributes two (semi-)automatic techniques to handle skew in a parallel UDO evaluation. With the result of this dissertation, users can spend their precious time on more important matters than optimizing code to deal with skew. We hope that this dissertation inspires much future research that leads to the age of "democratized" big data where any people from any background can easily process and analyze big data.

BIBLIOGRAPHY

[1] Yahoo! reaches for the stars with M45 supercomputing project. http://research.yahoo.com/node/1884.

[2] Deal with data. *Science*, 331(6018):639–806, 2011.

[3] Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. In *Proc. of the EDBT Conf.*, pages 99–110, 2010.

[4] Sameer Agarwal, Srikanth Kandula, Nicolas Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. Re-optimizing data-parallel computing. In *Proc. of the 9th NSDI Symp.*, 2012.

[5] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/.

[6] Amazon EC2 Spot Instances. http://aws.amazon.com/ec2/spot-instances/.

[7] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proc. of the 9th OSDI Symp.*, 2010.

[8] Mihael Ankerst, Markus M. Breunig, Hans peter Kriegel, and Jörg Sander. OPTICS: Ordering Points To Identify the Clustering Structure. In *Proc. of the SIGMOD Conf.*, pages 49–60, 1999.

[9] Apache Giraph Project. Apache Giraph. http://incubator.apache.org/giraph/, 2012.

[10] Apache Hadoop Project. Powered By Hadoop. http://wiki.apache.org/hadoop/PoweredBy/, 2011.

[11] Oceanic Remote Chemical Analyzer (ORCA). http://armbrustlab.ocean.washington.edu/.

[12] Teradata Aster. http://www.asterdata.com/.

[13] Sivnath Babu and Pedro Bizarro. Adaptive query processing in the looking glass. In *Proc. of the Second CIDR Conf.*, pages 238–249, 2005.

[14] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In *Proc. of the First SoCC Conf.*, pages 119–130, 2010.

[15] Jacek Becla and Kian-Tat Lim. Report from the SciDB meeting (a.k.a. extremely large database workshop). `http://xldb.slac.stanford.edu/download/attachments/4784226/sciDB2008_report.pdf`, 2008.

[16] Hernando Bedoya, Fredy Cruz, Daniel Lema, and Satid Singkorapoom. *Stored procedures, triggers, and user-defined functions on db2 universal database for iseries.* IBM Redbooks. IBM, 2006.

[17] M.J. Berger and S.H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *Computers, IEEE Transactions on*, C-36(5), 1987.

[18] Kevin S. Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Eltabakh, Carl-Christian Kanne, Fatma Ozcan, and Eugene J. Shekita. Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. *Proc. of the VLDB Endowment*, 4(12):1272–1283, 2011.

[19] Bigben. `http://www.psc.edu/machines/cray/xt3/`.

[20] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in MapReduce. In *Proc. of the SIGMOD Conf.*, pages 975–986, 2010.

[21] About the Blue Waters project. `http://www.ncsa.illinois.edu/BlueWaters/`.

[22] Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proc. of the 27th ICDE Conf.*, pages 1151–1162, 2011.

[23] Dhruba Borthakur. The Hadoop distributed file system: Architecture and design. `http://lucene.apache.org/hadoop/hdfs_design.pdf`, 2007.

[24] Jihad Boulos and Kinji Ono. Cost estimation of user-defined methods in object-relational database systems. *SIGMOD Record*, 28(3), 1999.

[25] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *Proc. of the 7th WWW Conf.*, pages 107–117, 1998.

[26] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. HaLoop: efficient iterative data processing on large clusters. *Proc. of the VLDB Endowment*, 3(1):285–296, 2010.

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

[28] Surajit Chaudhuri. What next?: a half-dozen data management research goals for big data and the cloud. In *Proc. of the PODS Conf.*, pages 1–4, 2012.

[29] Yanpei Chen, Sara Alspaugh, and Randy H. Katz. Design insights for MapReduce from diverse production workloads. Technical Report UCB/EECS-2012-17, EECS Department, University of California, Berkeley, 2012.

[30] Byung-Gon Chun, Ling Huang, Sangmin Lee, Petros Maniatis, and Mayur Naik. Mantis: Predicting system performance through program analysis and modeling. *CoRR*, abs/1010.0019, 2010.

[31] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce online. In *Proc. of the 7th NSDI Symp.*, 2010.

[32] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *Computing in Science and Engineering*, 5(1):46–55, 1998.

[33] M. Davis, G. Efstathiou, C. S. Frenk, and S. D. M. White. The evolution of large-scale structure in a universe dominated by cold dark matter. *Astroph. J.*, 292:371–394, May 1985.

[34] Brian Dawkins. Siobhan's problem: The coupon collector revisited. *The American Statistician*, 45(1):76–82, 1991.

[35] IBM InfoSphere Warehouse. `http://www.ibm.com/software/data/infosphere/warehouse/`.

[36] Jeffrey Dean. Evolution and future directions of large-scale storage and computation systems at google. In *Proc. of the First SoCC Conf.*, 2010.

[37] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. of the 6th OSDI Symp.*, 2004.

[38] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):139, 2007.

[39] Karen Devine, Erik Boman, Robert Heapby, Bruce Hendrickson, and Courtenay Vaughan. Zoltan data management service for parallel dynamic applications. *Computing in Science and Engg.*, 4(2), 2002.

[40] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE TKDE*, 2(1):44–62, 1990.

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

[42] David DeWitt, Jeffrey Naughton, Donovan Schneider, and S S. Seshadri. Practical skew handling in parallel joins. In *Proc. of the 18th VLDB Conf.*, pages 27–40, 1992.

[43] David J. DeWitt, Erik Paulson, Eric Robinson, Jeffrey Naughton, Joshua Royalty, Srinath Shankar, and Andrew Krioukov. Clustera: an integrated computation and data management system. *Proc. of the VLDB Endowment*, 1(1):28–41, 2008.

[44] eBay, Inc. `http://www.ebay.com/`.

[45] EMC. *Greenplum database 4.2 administrator guide*. EMC, 2012.

[46] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proc. of the 2nd KDD Conf.*, pages 226–231, 1996.

[47] Facebook, Inc. `http://www.facebook.com/`.

[48] Eric Friedman, Peter Pawlowski, and John Cieslewicz. Sql/mapreduce: a practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proc. of the VLDB Endowment*, 2(2):1402–1413, 2009.

[49] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[50] Jeffrey P. Gardner, Andrew Connolly, and Cameron McBride. Enabling knowledge discovery in a virtual universe. In *Proc. of the 2007 TeraGrid Symp.*, 2007.

[51] Jeffrey P. Gardner, Andrew Connolly, and Cameron McBride. Enabling rapid development of parallel tree search applications. In *Proc. of the 2007 CLADE Symp.*, 2007.

[52] Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a high-level dataflow system on top of map-reduce: the pig experience. *Proc. of the VLDB Endowment*, 2:1414–1425, August 2009.

[53] J. M. Gelb and E. Bertschinger. Cold dark matter. 1: The formation of dark halos. *Astroph. J.*, 436:467–490, 1994.

[54] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proc. of the 19th SOSP Symp.*, pages 29–43, 2003.

[55] Google. Freebase Data Dumps. `http://download.freebase.com/datadumps/`, 2010.

[56] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[57] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Min. Knowl. Discov.*, 1:29–53, January 1997.

[58] Greenplum database. `http://www.greenplum.com/`.

[59] Benjamin Gufler, Nikolaus Augsten, Angelika Reiser, and Alfons Kemper. Handling data skew in MapReduce. In *The First International Conference on Cloud Computing and Services Science*, 2011.

[60] Benjamin Gufler, Nikolaus Augsten, Angelika Reiser, and Alfons Kemper. Load balancing in mapreduce based on scalable cardinality estimates. In *Proc. of the 28th ICDE Conf.*, 2012.

[61] Hadoop. `http://hadoop.apache.org/`.

[62] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A self-tuning system for big data analytics. In *Proc. of the Fifth CIDR Conf.*, 2011.

[63] Michael A. Heroux, Padma Raghavan, and Horst D. Simon, editors. *Parallel Processing for Scientific Computing*, chapter 6. SIAM, 2006.

[64] Tony Hey, Stewart Tansley, and Kristin Tolle. *The Fourth Paradigm: Data-Intensive Scientific Discovery.* Microsoft, 2009.

[65] K.A Hua, Chiang Lee, and C.M Hua. Dynamic load balancing in multicomputer database systems using partition tuning. *IEEE TKDE*, 7(6):968–983, 1995.

[66] Kien Hua and Chiang Lee. Handling data skew in multiprocessor database computers using partition tuning. In *Proc. of the 17th VLDB Conf.*, 1991.

[67] Shadi Ibrahim, Hai Jin, Lu Lu, Song Wu, Bingsheng He, and Li Qi. LEEN: Locality/Fairness-Aware Key Partitioning for MapReduce in the Cloud. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 17–24, 2010.

[68] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proc. of the EuroSys Conf.*, pages 59–72, 2007.

[69] ISO/IEC 9075-*:2011. *Database Languages - SQL*. ISO, Geneva, Switzerland, 2011.

[70] Eaman Jahani, Michael J Cafarella, and Christopher Re. Automatic Optimization for MapReduce Programs. *Proc. of the VLDB Endowment*, 4(6):1–12, 2011.

[71] jeff kelly, david vellante, and david floyer. big data market size and vendor revenues. `http://wikibon.org/wiki/v/big_data_market_size_and_vendor_revenues`.

[72] Jimmy Lin. Cloud 9: A MapReduce library for Hadoop. `http://www.umiacs.umd.edu/~jimmylin/Cloud9/docs/index.html`, 2010.

[73] M.G. Kalyuzhnaya, D.A.C. Beck, and L. Chistoserdova. Functional metagenomics of methylotrophs. *Methods in Enzymology*, 495, 2011.

[74] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. PEGASUS: a peta-scale graph mining system implementation and observations. In *ICDM*, 2009.

[75] Soila Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. An analysis of traces from a production MapReduce cluster. In *CCGrid*, 2010.

[76] Qifa Ke, Vijayan Prabhakaran, Yinglian Xie, Yuan Yu, Jingyue Wu, and Junfeng Yang. Optimizing data partitioning for data-parallel computing. In *HotOS*, 2011.

[77] Masaru Kitsuregawa and Yasushi Ogawa. Bucket spreading parallel hash: a new, robust, parallel hash join method for data skew in the super database computer (sdc). In *Proc. of the 16th VLDB Conf.*, pages 210–221, 1990.

[78] Dan Klein and Christopher D. Manning. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1*, ACL '03, pages 423–430, 2003.

[79] S. R. Knollmann and A. Knebe. AHF: Amiga's Halo Finder. *Astroph. J. Suppl.*, 182:608–624, 2009.

[80] Paraschos Koutris and Dan Suciu. Parallel evaluation of conjunctive queries. In *Proc. of the PODS Conf.*, pages 223–234, 2011.

[81] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proc. of the First SoCC Conf.*, 2010.

[82] Yongchul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. SkewReduce: skew-resistance execution of a distributed job in Hadoop. `http://skewreduce.googlecode.com/`, 2011.

[83] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. SkewTune:. `http://skewreduce.googlecode.com/`, 2012.

[84] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. SkewTune: Mitigating Skew in MapReduce Applications. Technical Report UW-CSE-12-03-03, University of Washington, March 2012.

[85] YongChul Kwon, Dylan Nunley, Jeffrey P. Gardner, Magdalena Balazinska, Bill Howe, and Sarah Loebman. Scalable clustering algorithm for N-body simulations in a shared-nothing cluster. In *Proc. of the 22nd Scientific and Statistical Database Management Conference (SSDBM)*, 2010.

[86] Boduo Li, Edward Mazur, Yanlei Diao, Andrew McGregor, and Prashant Shenoy. A Platform for Scalable One-Pass Analytics using MapReduce. In *Proc. of the SIGMOD Conf.*, June 2011.

[87] Wei Li, Dengfeng Gao, and Richard Snodgrass. Skew handling techniques in sort-merge join. In *Proc. of the SIGMOD Conf.*, pages 169–180, 2002.

[88] Jeff W. Lichtman, R. Clay Reid, Hanspeter Pfister, and Michael F. Cohen. Discovering the wiring diagram of the brain. In *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft, 2009.

[89] Jimmy Lin. The Curse of Zipf and Limits to Parallelization: A Look at the Stragglers Problem in MapReduce. In *7th Workshop on Large-Scale Distributed Systems for Information Retrieval*, 2009.

[90] Jimmy Lin and Michael Schatz. Design patterns for efficient graph algorithms in mapreduce. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, pages 78–85, 2010.

[91] Yuting Lin, Divyakant Agrawal, Chun Chen, Beng Chin Ooi, and Sai Wu. Llama: leveraging columnar storage for scalable join processing in the mapreduce framework. In *Proc. of the SIGMOD Conf.*, pages 961–972, 2011.

[92] Steve Lohr. The age of big data. http://www.nytimes.com/2012/02/12/sunday-review/big-datas-impact-in-the-world.html, 2012.

[93] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. of the VLDB Endowment*, 5(8):716–727, 2012.

[94] Large Synoptic Survey Telescope. http://www.lsst.org/.

[95] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. of the SIGMOD Conf.*, pages 135–146, 2010.

[96] Ahmed Metwally and Christos Faloutsos. V-smart-join: a scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *Proc. of the VLDB Endowment*, 5(8):704–715, 2012.

[97] Microsoft. User-defined functions. http://msdn.microsoft.com/en-us/library/ms191007.aspx.

[98] Kristi Morton, Abram Friesen, Magdalena Balazinska, and Dan Grossman. Estimating the progress of MapReduce pipelines. In *Proc. of the 26th ICDE Conf.*, March 2010.

[99] Richard P. Mount. The office of science data-management challenge. Technical report, Department of Energy, 2004.

[100] N-Body Shop Group. ChaNGa massively parallel N-body code. http://www-hpcc.astro.washington.edu/tools/changa.html.

[101] Netezza, inc. http://www.netezza.com/.

[102] Alper Okcan and Mirek Riedewald. Processing theta-joins using mapreduce. In *Proc. of the SIGMOD Conf.*, pages 949–960, 2011.

[103] Leonid Oliker and Rupak Biswas. Plum: parallel load balancing for adaptive unstructured meshes. *J. Parallel Distrib. Comput.*, 52(2), 1998.

[104] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proc. of the SIGMOD Conf.*, pages 1099–1110, 2008.

[105] Oracle. http://www.oracle.com/database/.

[106] Oracle. Oracle database data cartridge developer's guide 11g release 2 (11.2). `http://docs.oracle.com/cd/e11882_01/appdev.112/e10765/toc.htm`, 2010.

[107] M. Parashar, H. Liu, Z. Li, V. Matossian, C. Schmidt, G. Zhang, and S. Hariri. AutoMate: Enabling Autonomic Applications on the Grid. *Cluster Computing*, 9(2), 2006.

[108] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel R. Madden, and Michael Stonebraker. A comparison of approaches to large scale data analysis. In *Proc. of the SIGMOD Conf.*, 2009.

[109] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4), 2005.

[110] PostgreSQL Global Development Group. Extending sql. `http://www.postgresql.org/docs/9.1/static/extend.html`, 2012.

[111] Xiaohong Qiu, Jaliya Ekanayake, Scott Beason, Thilina Gunarathne, Geoffrey Fox, Roger Barga, and Dennis Gannon. Cloud technologies for bioinformatics applications. In *MTAGS '09: Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*, pages 1–10, 2009.

[112] Sriram Rao, Raghu Ramakrishnan, Adam Silberstein, Mike Ovsiannikov, and Damian Reeves. Sailfish: A framework for large scale data processing. Technical Report YL-2012-002, Yahoo! Labs, 2012.

[113] D. Reed, J. Gardner, T. Quinn, J. Stadel, M. Fardal, G. Lake, and F. Governato. Evolution of the mass function of dark matter haloes. *Monthly Notices of the Royal Astronomical Society*, 346:565–572, December 2003.

[114] Kai Ren, YongChul Kwon, Magdalena Balazinska, and Bill Howe. Hadoop's adolescence: a comparative workload analysis from three research clusters. Technical Report UW-CSE-12-06-01, University of Washington, June 2012.

[115] Florin Rusu and Alin Dobra. Glade: a scalable framework for efficient analytics. *SIGOPS Oper. Syst. Rev.*, 46(1):12–18, 2012.

[116] Michael C. Schatz. CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics*, 25(11):1363–1369, June 2009.

[117] Ambuj Shatdal and Jeffrey Naughton. Adaptive parallel aggregation algorithms. In *Proc. of the SIGMOD Conf.*, 1995.

[118] Ambuj Shatdal and Jeffrey F. Naughton. Using shared virtual memory for parallel join processing. In *Proc. of the SIGMOD Conf.*, pages 119–128, 1993.

[119] An Overview of SKID. `http://www-hpcc.astro.washington.edu/tools/skid.html`.

[120] Marc Snir and Steve Otto. *MPI-The Complete Reference: The MPI Core*. The MIT Press, second edition, 1998.

[121] Christopher Southan and Graham Cameron. Beyond the tsunami: Developing the infrastucture to deal with life sciences data. In *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft, 2009.

[122] V. Springel, S. D. M. White, A. Jenkins, C. S. Frenk, N. Yoshida, L. Gao, J. Navarro, R. Thacker, D. Croton, J. Helly, J. A. Peacock, S. Cole, P. Thomas, H. Couchman, A. Evrard, J. Colberg, and F. Pearce. Simulations of the formation, evolution and clustering of galaxies and quasars. *NATURE*, 435:629–636, June 2005.

[123] SQL Server. `http://www.microsoft.com/sqlserver/`.

[124] Microsoft SQL Server 2008 R2 Parallel Data Warehouse. `http://www.microsoft.com/sqlserver/en/us/solutions-technologies/data-warehousing/pdw.aspx`.

[125] Joachim Gerhard Stadel. *Cosmological N-body simulations and their analysis*. PhD thesis, University of Washington, 2001.

[126] Alexander Szalay and Jim Gray. 2020 computing: Science in an exponential world. *Nature*, 440:413–414, 2006.

[127] The Mahout Team. Apache mahout project. `http://mahout.apache.org/`.

[128] Teradata. `http://www.teradata.com/`.

[129] the Hadepot team. Hadepot: repository of mapreduce applications. `http://nuage.cs.washington.edu/repository.php`, 2011.

[130] The Hive Team. Apache Hive. `http://hadoop.apache.org/hive/`.

[131] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. of the VLDB Endowment*, 2(2):1626–1629, 2009.

[132] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. Data warehousing and analytics infrastructure at facebook. In *Proc. of the SIGMOD Conf.*, pages 1013–1020, 2010.

[133] Twitter, Inc. `http://www.twitter.com/`.

[134] Prasang Upadhyaya, YongChul Kwon, and Magdalena Balazinska. A latency and fault-tolerance optimizer for online parallel query plans. In *Proc. of the SIGMOD Conf.*, June 2011.

[135] Rares Vernica, Andrey Balmin, Kevin S. Beyer, and Vuk Ercegovac. Adaptive MapReduce using situation-aware mappers. In *Proc. of the EDBT Conf.*, 2012.

[136] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using MapReduce. In *Proc. of the SIGMOD Conf.*, pages 495–506, 2010.

[137] Vertica, inc. `http://www.vertica.com/`.

[138] Christopher Walton, Alfred Dale, and Roy Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *Proc. of the 17th VLDB Conf.*, 1991.

[139] Keith Wiley, Andrew Connolly, Jeffrey P. Gardner, Simon Krughof, Magdalena Balazinska, Bill Howe, YongChul Kwon, and Yingyi Bu. Astronomy in the cloud: Using MapReduce for image coaddition. *Publications of the Astronomical Society of the Pacific (PASP)*, 123(901):366–380, March 2011.

[140] Windows Azure. `http://www.windowsazure.com/`.

[141] Worldwide LHC computing grid. `http://wlcg.web.cern.ch/`, 2012.

[142] Joel L. Wolf, Daniel M. Dias, and Philip S. Yu. An effective algorithm for parallelizing sort merge joins in the presence of data skew. In *Proceedings of the second international symposium on databases in parallel and distributed systems*, DPDS '90, pages 103–115, 1990.

[143] Joel L. Wolf, Daniel M. Dias, Philip S. Yu, and John Turek. An effective algorithm for parallelizing hash joins in the presence of data skew. In *Proc. of the 7th ICDE Conf.*, pages 200–209, 1991.

[144] Yu Xu and Pekka Kostamaa. Efficient outer join data skew handling in parallel dbms. *Proc. of the VLDB Endowment*, 2(2), 2009.

[145] Yu Xu, Pekka Kostamaa, Xin Zhou, and Liang Chen. Handling data skew in parallel joins in shared-nothing systems. In *Proc. of the SIGMOD Conf.*, pages 1043–1052, 2008.

[146] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *Proc. of the 22nd SOSP Symp.*, 2009.

[147] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proc. of the 8th OSDI Symp.*, 2008.

[148] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proc. of the 8th OSDI Symp.*, 2008.

[149] Chi Zhang, Feifei Li, and Jeffrey Jestes. Efficient parallel knn joins for large data in mapreduce. In *Proc. of the EDBT Conf.*, pages 38–49, 2012.

[150] Jiaxing Zhang, Hucheng Zhou, Rishan Chen, Xuepeng Fan, Zhenyu Guo, Haoxiang Lin, Jack Y. Li, Wei Lin, Jingren Zhou, and Lidong Zhou. Optimizing data shuffling in data-parallel computation by understanding user-defined functions. In *Proc. of the 9th NSDI Symp.*, 2012.

[151] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Zhiyong Xu. SJMR:Parallelizing Spatial Join with MapReduce on Clusters. In *CLUSTER*. IEEE, 2009.