RICE UNIVERSITY

Understanding and Improving the Efficiency of Failure Resilience for Big Data Frameworks

by

Florin Dinu

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

Dr. T. S. Eugene Ng (Chair),
Associate Professor,
Computer Science

Dr. Alan L. Cox,
Associate Professor,
Computer Science

Dr. Edward W. Knightly,
Professor,
Electrical and Computer Engineering

Dr. Christos Gkantsidis,
Researcher,
Microsoft Research Cambridge

HOUSTON, TEXAS
MAY, 2013
Understanding and Improving the Efficiency of Failure Resilience for Big Data Frameworks

Florin Dinu

Abstract

Big data processing frameworks (MapReduce, Hadoop, Dryad) are hugely popular today because they greatly simplify the deployment and execution of big data analysis jobs requiring the use of many machines in parallel. A strong selling point of these frameworks is their built-in failure resilience support. Big data frameworks can run computations to completion despite occasional failures in the system. However, an important but overlooked point has been the efficiency of their failure resilience. The vision of this thesis is that big data frameworks should not only be failure resilient but that they should provide the resilience in an efficient manner. This means both minimizing the impact of failures on computations as well as minimizing the cost of running proactive failure resilience algorithms during failure-free periods.

Towards the end goal of enabling efficient failure resilience for big data frameworks, this thesis makes two contributions. The first part of the thesis presents the first in-depth analysis of the efficiency of the failure resilience provided by Hadoop, the most popular big data processing framework today. The results show that even single machine failures can lead to large, variable and unpredictable job running times. This thesis discovers the causes behind this inefficient behavior, determines the responsible Hadoop mechanisms and points out their limitations. The second part of the thesis focuses on providing efficient failure resilience for the common case of computations comprised of multiple jobs. We present the design, implementation and evaluation of RCMP, a MapReduce system originating from the fundamental insight that using data replication to enable failure resilience oftentimes
leads to significant and unnecessary increases in computation running time. In contrast, RCMP is designed to use job re-computation as a first-order failure resilience strategy. Job re-computations under RCMP are efficient. Specifically, RCMP re-computes the minimum amount of work necessary and uniquely it ensures that this minimum re-computation work is performed efficiently. In particular, RCMP mitigates hot-spots that affect data transfers during job re-computations and also ensures that the available compute node parallelism is well exploited.
Acknowledgements

Most importantly, I would like to thank my adviser Prof. Eugene Ng for his mentorship, support and understanding during my PhD. I am deeply grateful to him because he went beyond simply giving me good advice. Instead, he was the first professor that took the time to help me improve by specifically addressing my personal way of thinking and viewing the world. I have learned a great deal from my adviser, things that I am happy to say are not limited to the professional level but that have helped me become a better person overall. I deeply admire his incredible patience and I can only hope to be able to repay him by showing the same patience to others. I am also grateful to him for teaching me how to be more efficient and for taking better care of my time than I would have taken myself.

I am very grateful to Dr. Alan Cox for his career advice and for taking the time to give me insightful research feedback so many times during my graduate studies. I thoroughly enjoyed collaborating with Dr. Cox on several projects.

A big thank you is due to Dr. Christos Gkantsidis, my mentor during an internship at Microsoft Research Cambridge. I am indebted to him for the opportunity to work in an environment which has changed my way of thinking about efficiency and being goal oriented. I am also very grateful to him for his mentorship, career advice, insightful feedback and for agreeing to be part of my thesis committee.

I would also like to thank Dr. Edward Knightly for his insightful comments on both my Master’s and PhD thesis.

I have greatly benefited from discussions with my group mates which have also been good friends: Bo Zhang, Guohui Wang, Zheng Cai, Jie Zheng, Yiting Xia, Xiaoye Sun, Ruiqi Liu, Apoorv Agarwal and Fred Liu.

I would like to deeply thank my parents for their unconditional love and support. I
cannot imagine how difficult it is to see your only child for a few days a year while he is living on the other side of the planet. I am deeply grateful for their understanding.

I would like to thank Dr. Mihaela Constantin for her love and support during the better part of this PhD. Looking back I also realize how extremely fortunate I was to have very good friends in Houston. I do not dare imagine how boring Houston would have been without them.
Contents

Abstract i
List of Illustrations viii
List of Tables xi

1 Introduction 1
1.1 Thesis Contributions .............................................. 3
1.2 Thesis Outline ......................................................... 5

2 Background 6
2.1 Why the Need for Big Data Frameworks? .......................... 6
2.2 MapReduce Background .............................................. 8
   2.2.1 Overview of the MapReduce Programming Model ............. 8
   2.2.2 The Major System Components ................................. 10
   2.2.3 Speculative Execution .......................................... 12
   2.2.4 The Difference Between Re-computation and Speculative Execution 12
   2.2.5 Replication of Reducer Output Data ........................... 13
2.3 Failure Model ......................................................... 13

3 Understanding the Effects and Implications of Compute Node Related Failures in Hadoop 15
3.1 Motivation and Outline of Analysis ................................ 15
3.2 Failures in Hadoop ................................................... 19
   3.2.1 How Hadoop Deals with TaskTracker Failures ................. 20
   3.2.2 How Hadoop Deals with DataNode Failures .................... 23
3.3 Experiments Exposing the Inefficiencies of Failure Detection and Recovery in Hadoop

3.3.1 Methodology

3.3.2 TaskTracker Failure Analysis

3.3.3 DataNode Failure Analysis - Delayed Speculative Execution

3.3.4 DataNode Failure Analysis - Not Sharing Failure Information

3.4 Discussions and Implications

3.5 Related Work

3.6 Chapter Summary

4 RCMP: A System Enabling Efficient Re-computation Based Failure Resilience for Big Data Analytics

4.1 Motivation and Outline of RCMP’s Benefits

4.2 Why Replication is Problematic

4.2.1 Overrated Benefits of Replication

4.2.2 Indirect Costs of Replication

4.3 Improving Re-computations with RCMP

4.3.1 Overall System Design

4.3.2 RCMP During Re-computation

4.3.3 System Building Challenges

4.4 Discussion

4.4.1 Storage Footprint and Management

4.4.2 RCMP Re-computes Tasks Entirely

4.4.3 RCMP and the Collocation of Storage and Computation

4.5 Evaluation

4.5.1 Methodology and Environment

4.5.2 Overall System Comparisons

4.5.3 Mitigating Hot-spots
Illustrations

2.1 Data flow in MapReduce. ........................................... 9
2.2 Key-value pair distribution between mapper and reducer tasks. .......... 10

3.1 Clusters of job running times under TaskTracker failure. Without any failures the average job running time is 220s. ......................... 26
3.2 Illustration of early notifications. The tuple format is (map name, time the penalty expires, $F^R_j(M)$). For example, (A,3,2) means at time 3 a new connection should be attempted to get mapper output A, and there have been 2 failed attempts so far. The tuple values are taken immediately after the corresponding timestamp. This example considers that notifications are sent when $F^R_j(M) = 5$. Note that this occurs at different moments, shown by rectangles. ........................................... 28
3.3 Effect of an increased number of reducers. .......................... 30
3.4 Single vs two concurrent jobs. ...................................... 31
3.5 Effect of RST packets. ................................................ 31
3.6 Illustration of delayed speculative execution. .......................... 33
3.7 Types of task progress rate spikes for speculated reducers. ............... 33
3.8 CDF of job running times for the 52 reducers, 1 wave case ............... 34
3.9 CDF of job running times for the 13 reducers, 1 wave case ............... 34
3.10 CDF of job running times for the 52 reducers, 4 waves case ............... 35
3.11 CDF of job running times for Figures 3.8, 3.9 and 3.10. ............... 35
3.12 LATE vs Hadoop CDF. ............................................. 40
3.13 Job running times for LATE using 52 reducers in 1 wave. . . . . . . . . . . 40
3.14 Job start-up times. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
3.15 CDF of job start-up time. . . . . . . . . . . . . . . . . . . . . . . . . . . . 43

4.1 The set of tasks and data transfers (both in bold) that are part of the re-computation of a MapReduce job under RCMP. M = mapper task, R = reducer task. A failure occurs just before Job 2 completes. The outputs of tasks $M_{11}$, $R_{11}$, $M_{21}$, $R_{21}$ are lost due to the failure and need to be re-computed. In contrast, existing systems re-compute everything. . . . . . 52
4.2 CDF of number of new failures reported per day for the STIC and SUG@R clusters at Rice University. . . . . . . . . . . . . . . . . . . . . . 56
4.3 RCMP system overview. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
4.4 Maximizing resource use for re-computation using reducer splitting. . . . . . 62
4.5 Increase contention (hot-spots) on storage during re-computation. . . . . . 64
4.6 The danger of re-using mapper outputs incorrectly when using reducer splitting. The highlighted tasks are re-computed. . . . . . . . . . . . . . . 68
4.7 Illustration of a potential problem caused by hash partitioning and reducer splitting. Highlighted tasks are re-computed. . . . . . . . . . . . . . . 69
4.8 Illustration of how mapper locality can influence how far re-computations cascade back. The large red arrow points to the furthest point to which re-computation will cascade to under a single node failure. Tasks affected by failure are highlighted. Node X runs tasks *.X. . . . . . . . . . . . . . . 73
4.9 The different moments at which failures are injected and their effects on the 7-job computation when using RCMP. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
4.10 Hadoop vs RCMP under single and no failures, 1 mapper slot and 1 reducer slot per node. Lower is better. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
4.11 Hadoop vs RCMP under single and no failures, 2 mapper slots and 2 reducer slots per node. Lower is better. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
4.12 Hadoop vs RCMP under double failures, 1 mapper slot and 1 reducer slot per node. Lower is better.

4.13 Splitting reducers mitigates hot-spots and accelerates mappers from the next job. CDF based on the experiments from Figure 4.10 (1 mapper slot and 1 reducer slot per node) when failures are injected at job 7 for RCMP (last two bars to the right). Lower x-axis values are better.

4.14 Splitting reducers mitigates hot-spots and accelerates mappers from the next job. CDF based on the experiments from Figure 4.11 (2 mapper slots and 2 reducer slots per node) when failures are injected at job 7 for RCMP (last two bars to the right). Lower x-axis values are better.

4.15 CDF of re-computation speed-up for RCMP for the experiment in Figure 4.10. 1 mapper slot and 1 reducer slot per node. Bigger x-axis values are better.

4.16 CDF of re-computation speed-up for RCMP for the experiment in Figure 4.11. 2 mapper slots and 2 reducer slots per node. Bigger x-axis values are better.

4.17 CDF of reducer running time for the experiment in Figure 4.10 when the failure is injected at job 7 for RCMP (last two bars to the right). 1 mapper slot and 1 reducer slot per node. Lower x-axis values are better.

4.18 CDF of reducer running time for the experiment in Figure 4.11 when the failure is injected at job 7 for RCMP (last two bars to the right). 2 mapper slots and 2 reducer slots per node. Lower x-axis values are better.

4.19 Efficiently using all available resources for re-computation with reducer splitting.

4.20 Speed-up obtained with RCMP from having fewer reducers waves during re-computation compared to the initial run.

4.21 Speed-up obtained with RCMP from having fewer mapper waves during re-computation compared to the initial run.
3.1 Variables used for failure handling in Hadoop. The format is $X_j^R(M)$. A subscript denotes the variable is per job. A superscript denotes the variable is per reducer. The parenthesis denotes that the variable applies to a mapper.
Chapter 1

Introduction

The present is increasingly data driven. Recent years have seen a dramatic diversification in the number of leveraged data sources and an equally significant increase in their size. The trend is only expected to continue and intensify. The result of leveraging this so called big data has been a spur in innovation, productivity and revenue. For example, companies have understood the importance of leveraging data generated from user activity on the Internet (click-streams, searches, comments, location and connectivity information, media uploads) for business decisions. In 2012 Facebook was collecting 500TB of user-related data per day [7] while in 2008 Google was already processing more than 20PB of data per day [43]. In this context, it is of great importance to process the available data in a fast, efficient and scalable fashion.

Processing large amounts of data is not a fundamentally new undertaking. It has long been necessary for scientific computing or for managing internal company records. Well known tools have been developed and have long existed for these cases. For instance, programs written on top of the Message Passing Interface (MPI [15]) library have been used for scientific computing while Relational Database Management Systems (RDBMS [19]) have been the predominant choice for managing the needs of a company. Unfortunately, the sheer size of today’s datasets as well as the specific way in which they need to be processed proved to be problematic for existing systems. For instance, current RDBMS offerings only scale to tens or at most hundreds of compute nodes [23]. RDBMSes also can take significant time to install and configure properly in order to provide reasonable performance [84] thus impacting overall productivity. The relational data model is not very well suited for unstructured or semi-structured data which is often generated over the web.
Moreover, the failure resilience provided by RDBMSes can be unsatisfactory as it is often based on the assumption that failures are a very rare event [23]. Similarly, MPI-based programs are difficult to write and debug because the programmer needs to deal with the details of the distributed communication and synchronization between nodes. Also, for failure resilience the programmer needs to implement the functionality necessary to detect and deal with failures.

As a consequence of these limitations of previous systems, Internet companies have proposed a number of new systems deemed big data processing frameworks. These frameworks are built from the ground up to support processing big datasets in a simple and highly scalable fashion. These frameworks are based on the insight that many applications today are easily parallelizable and this fits ideally with the currently viable economic trend of building large scale clusters of shared-nothing commodity machines. These frameworks also greatly simplify the programming experience by using a simple programming model and by abstracting away many complex details concerning computation management and data communication between machines. Today, several large Internet companies as well as a multitude of small companies have built their infrastructure and systems around such frameworks [85]. The design details of the first big data processing framework called MapReduce was published by Google in 2004 [42]. Subsequently, this design served as the basis for Apache Hadoop [9], an open source alternative which has become the most popular big data framework.

A critically important selling point of these new big data frameworks is their declared ability to continue to make progress and eventually finish the computation despite failures affecting the system. This is important because increasing the number of machines used for a computation also increases the probability that a failure will adversely impact the computation.

The goal of this thesis is to enable efficient failure resilience for big data frameworks. By efficient we mean minimizing the impact of failures on computations as well as minimizing the cost of running proactive failure resilience algorithms during failure-free pe-
Towards this goal we aim to both understand the behavior and limitations of state-of-the-art systems and also to develop new, more efficient solutions for providing failure resilience.

While the probability of a big data analytics job being impacted by a failure as well as the magnitude of the impact will naturally vary depending on the computation performed, its size, configuration and properties of the environment, we believe that understanding how to efficiently deal with failures is always important. There are several reasons for this. First, most big data analysis jobs today run on economically viable clusters of commodity components and therefore failures cannot be completely ignored [41, 89, 8]. Second, efficient failure resilience means that users can benefit from a more predictable experience. They will not have to pay unnecessarily high monetary costs because of failures and the waiting times for job results will never be unnecessarily inflated.

1.1 Thesis Contributions

Specifically, this thesis makes two important contributions.

**Effects and implications of compute node related failures in Hadoop** We start by understanding the behavior of a state-of-the-art big data framework. We present the first analysis of the efficiency of the failure resilience provided by Hadoop, the most popular big data framework today. In particular, we analyze Hadoop’s behavior under failures involving compute nodes. Our findings show significant shortcomings in Hadoop: even a single failure can result in inflated, variable and unpredictable job running times, all undesirable properties in a distributed system.

We identify the causes behind this distressing state-of-the-art and point out the responsible Hadoop mechanisms and their limitations. A first cause relates to Hadoop’s speculative execution algorithm which is responsible for handling slow tasks by duplicating or restarting them. Under failures, the algorithm can cease to detect slow tasks reliably and this significantly impacts overall job running times. The reason is that the algorithm is statistical in nature and the statistics are skewed by fast advancing tasks that can appear because of
failures. A second cause is that Hadoop uses connection failures between tasks to infer task failures. However, Hadoop does not have enough visibility into the infrastructure to detect whether the problem is at the source, at the destination or in the network. As a result, false positives appear in the system: otherwise localized failures propagate to healthy tasks. A third cause is that the efficiency of failure detection in Hadoop is a function of the current state of the computation. This results in variable failure detection times. Fourth, failures are re-discovered individually by each task at the cost of great degradation in job running time. The reason is that Hadoop focuses on extreme scalability and thus sacrifices potential improvements resulting from sharing failure information between tasks.

We believe that our analysis can positively influence the design of several core components of Hadoop and of similar big data frameworks. Our analysis suggests the need for devising more robust speculative execution algorithms that are resilient to skew in task progress rates, potentially completely forgoing comparisons between tasks when making decisions. Our analysis also reveals several decisions that Hadoop makes based on scarce information about the environment. While good for simplicity we show the negative effects of those decisions. This suggests the need for more elaborate methods of collecting environmental information [64] that can lead to quicker and more accurate decisions.

**RCMP: Enabling efficient re-computation based failure resilience for big data analytics** Our second contribution looks at the case of multi-job I/O-intensive computations. Unfortunately, these can suffer a significant performance hit due to the fact that big data frameworks rely on data replication as the main failure resilience strategy. Replication is essentially an expensive operation for big data analytics because it puts the network and the storage under significant stress. Avoiding replication exposes the computation to the negative effects of failures which can cause data loss and trigger cascading re-computations. Unfortunately, today’s big data processing systems have no built-in support to handle cascading re-computations efficiently. They entirely re-compute all affected jobs.

We propose RCMP, a MapReduce system that uses job re-computation as a first-order failure resilience strategy. Re-computations in RCMP are efficient when failures do oc-
cur while bearing no cost during failure-free periods. RCMP persists across jobs mapper and reducer outputs that are materialized as part of successfully completed intermediate jobs. RCMP benefits from the fact that most failures usually affect only a small portion of the persisted data, allowing the bulk of it to be used to speed-up re-computation. On failures that cause data loss, RCMP decides which jobs must be re-computed and based on the persisted data it also determines the minimum number of tasks that need to be re-computed for each job. Unique to RCMP are advanced capabilities designed to ensure that this minimum re-computation work is performed efficiently. Specifically, RCMP addresses two challenges that we discovered to appear in non-replicated MapReduce systems when re-computation is performed. First, RCMP can efficiently utilize the available compute node parallelism during re-computation by switching to a more fine-grained task scheduling granularity. Second, RCMP mitigates hot-spots that affect task data transfers during re-computations. Compared to RCMP, in our experiments, data replication is 30% to 100% worse during failure-free periods. More importantly, RCMP yields comparable or even better total multi-job running time even under single and double data loss events.

1.2 Thesis Outline

The rest of the thesis is organized as follows. Chapter §2 presents relevant MapReduce and Hadoop background. Chapter §3 details the first contribution, namely the analysis of the efficiency of failure resilience in Hadoop. Chapter §4 details the motivation, design and evaluation behind RCMP, our MapReduce system that uses efficient job re-computation as a primary failure resilience strategy. Chapter §5 discusses future work ideas and chapter §6 concludes. Related work can be found in chapter §3 and chapter §4.
Chapter 2

Background

2.1 Why the Need for Big Data Frameworks?

It has long been necessary to process large amounts of data. The High Performance Computing (HPC) community has been for decades involved in large computations for scientific purposes ranging from weather prediction and cosmological simulations to particle physics experiments. The tool of choice for this scenario has been writing parallel programs by extending the MPI library [15] which provides synchronization, and communication functionality between a set of processes which commonly run on different processors. Separately, companies have long needed to store and access large amounts of data for financial and administrative reasons. RDBMSes [19] have been by far the most popular tool for this latter scenario. Given the existence of these established options, the natural question to ask is why is there currently a need for new big data processing systems and why existing solutions cannot be directly leveraged? While there is certainly some overlap between what these solutions can do, and existing solutions do have a number of appealing qualities, none provides the combination of high productivity, scalability and failure resilience that is required today to make the best data-driven decisions.

There are two aspects to the need for high productivity. First, the system needs to be easily deployable and relatively easy to configure and manage in order to obtain reasonable performance. Unfortunately, RDBMSes can take significant time to install and configure properly. Researchers report obtaining dismal performance on some RDBMSes [84] and having to resort to expert advice from the vendor. On the other hand, big data frameworks have been designed to work out-of-the-box at reasonable performance levels with minimal configuration effort. The second aspect to high productivity is simplifying programming
experience. Writing distributed parallel programs is notoriously difficult and error-prone because the communication and the synchronization between machines need to be correctly implemented. While MPI provides communication and synchronization primitives, the burden remains on the programmer to implement the desired functionality correctly. Big data frameworks have addressed this shortcoming by providing a simple and slightly restrictive programming model. The user only needs to write serial functions (no parallel code) and the system takes care of all the communication, synchronization and management details required to perform the computation in parallel on many machines. This is a powerful idea that enables rapid deployment and prototyping. Programmers with no previous experience in writing parallel programs can now take advantage of a large scale distributed system.

Scalability is also important for making the best data-driven decisions. As datasets grow larger and larger, the time necessary to make important data-driven decisions remains small or decreases. A straightforward solution is to use as many as thousands or tens of thousands of machines in parallel. In this respect, current RDBMS offerings falls short as they can only be deployed at scales of tens or at most hundreds of compute nodes [23]. RDBMS deployment at larger scale, while not fundamentally impossible will likely be slowed by engineering challenges which have not been encountered at smaller scales [23].

Finally, the failure resilience provided by both RDBMSes as well as MPI programs is unsatisfactory. RDBMSes are typically based on the assumption that failures are a very rare event [23]. As a result, a common approach for RDBMSes is to restart a query if a failure is encountered. For MPI programs, the programmer needs to implement the functionality necessary to detect and deal with failures.

It is also worth mentioning other disadvantages of using MPI-based programs or RDBMSes. For instance, studies note that large clusters are often heterogeneous [97, 79] and RDBMSes are typically designed under the assumption that all nodes are similar [23]. Also, RDBMSes leverage the relational data model. Unfortunately, the data generated on the Internet is often semi-structured or unstructured. Processing this data with a RDBMS requires an expensive data loading phase [23, 84]. In contrast, big data frameworks provide
the flexibility of processing this data directly.

2.2 MapReduce Background

The MapReduce programming model and system design have been first proposed by Google in their seminal 2004 publication [42]. A major reason for the tremendous popularity that MapReduce has since enjoyed lies with the existence of open source implementations of the model. By far the most popular MapReduce implementation is Hadoop [9, 93] developed initially by Doug Cutting and Mike Cafarella in 2005. Since then, Hadoop has become a core Apache project and a multitude of companies have began using Hadoop or are offering improved Hadoop distributions [2]. Google has also proposed the Google File System (GFS) [53] a highly scalable distributed file system capable of serving large MapReduce computations. The Hadoop Distributed File System (HDFS) [83] is an open source implementation based on Google’s GFS design.

2.2.1 Overview of the MapReduce Programming Model

A MapReduce job is a unit of work that the client wants to be performed. It is a transformation of the job input. More descriptively, a MapReduce job is the application of two user defined functions (UDFs) to the job input data. A map UDF that is applied to the job input and a reduce UDF that is applied to the output of the map UDF. Commonly, multiple jobs can be linked together in workflows (typically DAGs or chains) in order to solve more complex problems. The MapReduce framework takes care of efficiently applying the map and reduce UDFs to the data in a distributed fashion by leveraging many compute nodes in parallel. For this, it separates the execution of a UDF into multiple tasks where each task can be assigned to a different compute node and is responsible for processing a disjoint subset of the input. There are two types of tasks in MapReduce: mappers and reducers. Mappers apply the map UDF and the reducers apply the reduce UDF. Importantly, the map and the reduce UDFs are serial functions. This greatly simplifies the job of the programmer which does not need to implement any parallel logic. The parallelism is generated by the
system which takes care of applying the serial map and reduce UDFs to the data on many machines at the same time.

The input and output of a job are composed of key-value records and are stored in the distributed file system. Mapper tasks take as input a number of key-value records and apply the map UDF to each of the key-value records independently. On the other hand, a reducer task applies the reduce UDF to all key-value records in the mapper outputs that share the same key. Each reducer is usually responsible for a set of keys. Since the reducers process the output of the mappers, there is an exchange of data between the mapper and the reducer tasks. This exchange of data is called the shuffle phase. The purpose of the shuffle phase is to re-organize (it does not modify the data) the mapper output across the machines used such that it conforms to the specification of the reduce phase input. During the shuffle phase, reducers copy from mappers the key-value records that correspond to the keys they need to process. In practice, each reducer usually ends up getting part of each mapper’s output data. This results in an all-to-all traffic pattern between the nodes running the tasks.
Figure 2.1 depicts the MapReduce tasks and the exchange of data between them.

The system also has a method of deciding which keys will be assigned for processing to each reducer. A popular method is to use hash partitioning. Consider that each reducer has a numerical ID starting with 0. Hash partitioning works by taking each mapper output record, hashing its key and then performing on that hash a modulo operation using the total number of reducers. Subsequently, all records that yielded the same modulo result go to the same reducer. The process is illustrated in Figure 2.2 which for simplicity assumes that the result of hashing a key is the key itself.

2.2.2 The Major System Components

The system level components described in this subsection are based on Hadoop and HDFS. At the high level they are similar to the initial description of Google’s MapReduce, but the MapReduce implementation is proprietary so detailed information about it is not available.

A first major Hadoop system component is the JobTracker. This is a centralized component that manages jobs, schedules tasks and detects failures. A TaskTracker process runs on every compute node and is responsible for starting and managing tasks locally. TaskTrackers are configured with a number of mapper and reducer slots, the same number for
every TaskTracker. The mapper and the reducer slots are different, that is, a reducer cannot use a mapper slot and vice-versa. If a job requires more reducers (or mappers) than the number of reducer (or mapper) slots in the system, then the reducers (or mappers) are said to run in multiple waves. As an example, if the TaskTracker is configured with two reducer slots then only two reducers can concurrently run on that compute node. If that compute node needs to run a total of four reducers for some job then they will run in two waves, with a reducer in the second wave starting only after one reducer in the first wave finished. A TaskTracker periodically exchanges both liveness as well as task-related information with the JobTracker.

HDFS is composed of a centralized NameNode and of distributed DataNodes running on compute nodes. The NameNode is a centralized component that handles the file system metadata, decides on the block placement and responds to client requests involving the use or modification of metadata. DataNodes handle the read and write operations to the HDFS. A write timeout (WTO) or a read timeout (RTO) occurs when a HDFS write/read operation is interrupted by a DataNode failure. WTOs occur for reducers while RTOs occur for mappers. Connect timeouts (CTOs) can occur for both mappers and reducers, when they cannot connect to a DataNode.

For some of the discussions in this thesis the distinction between the NameNode and JobTracker is not important. All that matters is that there is a centralized component that manages computation and file system metadata. For these discussions, to simplify exposition, we may use the term Master to refer to the combined functionality of the NameNode and the JobTracker.

Each task is assigned a progress score which attempts to capture how close the task is to completion. A score of 0 means that the task has not yet begun while a score of 1 means that the task has completed. The progress score of a mapper is directly proportional to the percentage of its input that it has processed. For a reducer, a score of 0.33 signifies the end of the copy (shuffle) phase. At 0.33 all mapper outputs have been copied. A score of 0.66 signifies the end of the sort phase. Between 0.66 and 1 the reduce UDF is applied and the
output data is written to HDFS. Using the progress score one can define a progress rate, which is the ratio of the progress score over the current task running time. For example, it can take a task 15s to reach a score of 0.45, for a progress rate of 0.03/s.

2.2.3 Speculative Execution

The JobTracker runs a speculative execution algorithm which attempts to improve job running time by detecting under-performing tasks and duplicating them on another node in the hope that there they will progress faster. The algorithm in Hadoop 0.21.0 is a variant of the LATE algorithm [97]. Both algorithms rely on task progress rates. Both select a set of candidate under-performing tasks for speculation and then execute the candidate task that is estimated to finish farthest in the future. The difference lies in the method used to select the candidates. Hadoop takes a statistical approach. A candidate for speculation is a task whose progress rate is slower by at least one standard deviation than the average progress rate of all started tasks of the same kind (i.e. mapper or reducer) that belong to one job. Let \( Z(T_i) \) be the progress rate of a task \( T_i \) and \( T_{set} \) the set of all running or completed tasks of the same kind. A task \( T_{cur} \) can be speculatively executed if:

\[
\text{avg}(Z(T_i)_{T_i \in T_{set}}) - \text{std}(Z(T_i)_{T_i \in T_{set}}) > Z(T_{cur})
\] (2.1)

Intuitively, Hadoop speculates an under-performing task only when large variations in progress rates occur. In contrast, LATE attempts to speculate tasks as early as possible. For LATE, the candidates are the tasks with the progress rate below a SlowTaskThreshold, which is a percentile of the progress rates for a specific task type. Both algorithms speculate a task only after it has ran for at least 60s. To minimize the impact on available resources both algorithms cap the number of active speculative instances of each task to 1.

2.2.4 The Difference Between Re-computation and Speculative Execution

It is important to clarify how the notion of re-computation applies to MapReduce. The second contribution of this thesis, the RCMP system, focuses on job-level re-computation.
Such re-computation is necessary when the currently running job cannot continue to completion because part of its input data became unavailable. Its input needs to be re-generated and this is done by re-computing parts of a number of previous jobs. This should not be confused with the speculative execution mechanism. Speculative execution is a task-level mechanism and is useful only when the input to the job is available.

2.2.5 Replication of Reducer Output Data

In this thesis when we use the term replication we refer to data replication, that is, writing multiple copies of the same data on multiple nodes in the distributed file system.

Input and output files stored in the distributed file system are divided into blocks. HDFS uses a default block size of 64MB, though 256MB is also commonly encountered. Each reducer output may be comprised of several blocks and each block is a set of key-value records. The replication of reducer output data is based on a configuration parameter called a replication factor. As an example, a replication factor of 3 means that in total 3 copies of each reducer’s output block will exist in the distributed file system. Reducer output write operations are pipelined to minimize the amount of redundant data transmitted over the network. For example, if data is stored on node A and needs to be replicated on nodes B and C, then, in a pipelined write, data flows from node A to B and from node B to C.

2.3 Failure Model

In this work we focus on failures which from the point of view of the big data frameworks are fail-stop resource failures. This encompasses several common problems having various root causes. For example, a switch hardware failure, an end-host hardware failure or an unplanned maintenance episode are likely all interpreted by the big data framework as fail-stop compute node failures. Note also that any temporary failure lasting more than a threshold time will be treated by the big data framework as a fail-stop failure. The temporarily inaccessible resource may later return but by the time this happens the computation may have already accounted for the failure and the return of the resource provides no bene-
fit. Thus, our fail-stop resource failure model also handles certain temporary problems, for example transient node overload episodes that leave the node unresponsive for a prolonged period of time.

In this thesis we are not concerned with the root cause analysis of these problems. Our focus is designing big data frameworks that provide efficient failure resilience to guard against these problems. We aim to understand the effect that these problems have on big data applications. We also aim to understand the current mechanisms used by big data frameworks to react to failures, their interactions and limitations. Lastly, we aim to propose improved methods for providing efficient failure resilience.

In this thesis, "fail-stop resource failures" refers specifically to two types of resources. The first type are resources that facilitate access to job input and job output data. Such resources can be disk drives that store the data, compute nodes that have disk drives or big data framework OS processes that handle calls to access the data (e.g. the DataNode processes in Hadoop that handle read and write file system calls). The second type of resource are computing resources such as compute nodes or big data framework OS processes that perform the computation or manage processes that perform the computation (e.g. the TaskTracker processes in Hadoop).
Chapter 3

Understanding the Effects and Implications of Compute Node Related Failures in Hadoop

This chapter presents the first analysis of Hadoop’s efficiency under compute node failures. We find that even a single failure can result in inflated, variable and unpredictable job running times, all undesirable properties in a distributed system. We systematically track the causes underlying this distressing behavior and discuss the mechanisms that are involved, their interactions and limitations.

3.1 Motivation and Outline of Analysis

Hadoop has become a cloud workhorse [11]. Major Internet companies rely on Hadoop for their everyday needs involving extremely large datasets [85, 87, 13]: management tasks involving log processing [29, 12], business intelligence applications or the deployment of new products and platforms [33, 87]. Cloud service providers have added Hadoop to their list of offerings [14, 5]. Scientists use Hadoop for a wide range of purposes. To name only a few examples, Hadoop facilitated the implementation of scalable solutions and algorithms for data-intensive text processing [4], assembly of large genomes [6], graph mining [17], machine learning and data mining [3] and large scale social network analysis [22]. Hadoop has also received much attention from the research community. Several studies propose performance improvements [46, 60] or extensions to Hadoop: running Hadoop on a wider range of job types [35, 39], in more challenging environments [97], as a back-end in other large scale systems [87, 70] or as part of a hybrid architecture [23]. As a result of its widespread use and of the critical nature of the applications running on top of it, ensuring good performance for Hadoop jobs is essential.
In this chapter, we focus on Hadoop’s behavior under compute node failures. Little research work has been done on analyzing Hadoop’s performance under failures and understanding the efficiency of its design decisions in the context of failures. Given Hadoop’s popularity and the fact that at scale failures cannot be ignored this research direction has immediate practical relevance. We are the first to provide a thorough analysis of Hadoop under failures. The problem of dealing with failures is complex and our goal is to provide a deep and insightful technical analysis. We view our work as a necessary first step for solving what has become a chronically overlooked aspect: designing and building more robust failure detection and recovery algorithms for Hadoop. To this end, a collaborative effort from the community is needed: failure characteristics, job characteristics and cloud resource occupancy in real deployments need to be analyzed. Thus, in addition to the practical relevance, this research direction is rich in avenues for impactful future work.

Specifically, in this chapter we analyze Hadoop’s behavior under fail-stop failures of entire compute nodes and under fail-stop failures of the Hadoop components running on compute nodes (TaskTracker and DataNode). DataNode failures are important because they affect the availability of job input and output data and also delay read and write data operations which are central to Hadoop’s performance. TaskTracker failures are equally important because they affect running tasks as well as the availability of intermediate data (i.e. mapper outputs). Unlike failures affecting logically centralized Hadoop components (JobTracker and NameNode) which can be addressed by distributed coordination mechanisms [73], compute node failures require Hadoop to take explicit measures for detection and recovery and are more likely to cause subtle interactions with the environment or between Hadoop’s components.

Our measurements point to a real need for improvement. Surprisingly, we discover that a single failure can lead to large, variable and unpredictable job running times. For example, the running time of a job that takes 220s without failures can vary from 220s to as much as 1000s under TaskTracker failures and to 700s under DataNode failures. This is especially important for short jobs which are frequently encountered in the cloud and have
been shown to be a major use case for Hadoop and for big data analytics [97, 42, 38, 48]. Such large performance variations are detrimental. They cause unpredictable user costs and prolonged waiting for results, decrease overall cloud utilization and complicate scheduling. In our experiments on a predictable data center environment, the primary cause for the performance variations under failure was not the unavoidable penalty of having to restart and re-compute the tasks on the failed node. Instead, it was caused by inefficiencies in Hadoop’s failure detection and recovery algorithms.

We expose several such inefficiencies. First, Hadoop makes unrealistic assumptions about task progress rates. Hadoop seems to think that, with the exception of a few under-performing outliers, tasks progress at comparable rates. For Hadoop this warrants the use of a statistical speculative execution algorithm centered around average progress rates. Unfortunately, Hadoop’s assumption can be easily invalidated in practice. Both the cloud environment as well as other Hadoop design decisions can lead to very fast progressing tasks. For example, a number of recent proposals for improved cloud network design [62, 90] advocate accelerating specific network paths. Alternatively, imbalanced computations can lead to reducer tasks which are very fast because they process little data. Also, in a Hadoop job with multiple reducer waves, reducer tasks not belonging to the first wave can progress at very high rates because they do not have to wait for their input. We show that when Hadoop’s assumption about task progress rates is invalidated, a negative effect which we call delayed speculative execution can appear. This consists in one speculative execution decision severely delaying or even precluding subsequent speculative executions at great overall costs for the job running time.

Second, Hadoop trades off possible improvements resulting from communication between tasks for extreme scalability. Therefore, each task performs failure detection and recovery on its own. The unfortunate effect of this lack of sharing failure information is that multiple tasks could be left wasting time re-discovering a failure that has already been identified by another task. Moreover, a speculated task may have to re-discover the same failure that hindered the progress of the original task in the first place. We find that both
Hadoop’s speculative execution algorithm as well as the LATE algorithm [97] can be significantly impacted by failures. Importantly, these findings suggest that even state-of-the-art approaches to cause-aware speculative execution [27] may be insufficient. This is because a good speculative execution decision can be invalidated at runtime when the speculated task is affected by a failure. To ensure that speculated tasks help improve job running time, failure information needs to be effectively shared between tasks at job runtime.

Third, Hadoop uses connection failures between its tasks as a heuristic for detecting node failures. In part, this is warranted by the limited visibility that a cloud application can obtain about the cloud environment. Unfortunately, several factors can cause connection failures without implying node failures. Temporary overload conditions such as network congestion or excessive end-host load can cause connection failures. All these conditions are common in data centers [31, 41]. As a result, from only the news of a connection failure, Hadoop cannot reliably distinguish an underlying cause. We show that this limitation unnecessarily introduces false positives into the system. Specifically, otherwise localized failures involving a compute node can propagate to tasks running on healthy nodes. We call this the *induced reducer death* problem.

Fourth, the efficiency of failure detection in Hadoop is a function of the state of the computation at the time of the failure because the task-based failure detection mechanism in Hadoop relies on the currently running tasks and the data transfers between them. As the computation progresses, the state of the computation changes and with it the efficiency of the failure detection. The result are variable and unpredictable failure detection times. As we shall describe, Hadoop has actually two mechanisms for failure detection, one that is task based and one that is process based. The variable efficiency of the task-based mechanism means that each of the two mechanisms can end up reacting to a failure first, thus further increasing the variability and unpredictability of failure detection.

We believe that the points we highlight about the design decisions and the subtle interactions in a real-life system are important for understanding how to build more robust, efficient and performant big data frameworks. In practice, these subtle interactions can
easily invalidate benefits obtainable through smart solutions built on top. See for example the unexpected but serious ways in which failures affect the LATE and Hadoop speculative execution algorithms.

<table>
<thead>
<tr>
<th>Var.</th>
<th>Description</th>
<th>Var.</th>
<th>Description</th>
<th>Var.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P^R_j$</td>
<td>Time from reducer R’s start until it last made progress</td>
<td>$R_j$</td>
<td>Nr. of reducers currently running</td>
<td>$T^R_j$</td>
<td>Time since reducer R last made progress</td>
</tr>
<tr>
<td>$M_j$</td>
<td>Nr. of mappers (input splits) for a job</td>
<td>$D^R_j$</td>
<td>Nr. of mapper outputs copied by reducer R</td>
<td>$S^R_j$</td>
<td>Nr. of mappers that reducer R failed to shuffle from</td>
</tr>
<tr>
<td>$F^R_j(M)$</td>
<td>Nr. of times reducer R failed to copy mapper M’s output</td>
<td>$A^R_j$</td>
<td>Total nr. of shuffles attempted by reducer R</td>
<td>$Q_j$</td>
<td>Maximum running time among completed mappers</td>
</tr>
<tr>
<td>$N_j(M)$</td>
<td>Nr. of notifications that mapper M’s output is unavailable</td>
<td>$K^R_j$</td>
<td>Nr. of failed shuffle attempts by reducer R</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Variables used for failure handling in Hadoop. The format is $X^R_j(M)$. A subscript denotes the variable is per job. A superscript denotes the variable is per reducer. The parenthesis denotes that the variable applies to a mapper.

### 3.2 Failures in Hadoop

In this section we describe the mechanisms that Hadoop uses to guard against failures. Alongside the speculative execution algorithm described in subsection §2.2.3 these mechanisms cause the serious inefficiencies that we uncover in this chapter.

We identified these mechanisms by performing source code analysis on Hadoop version 0.21.0. The experiments in the rest of the chapter are also based on 0.21.0. At the beginning of November 2011 when we started the work on this project, version 0.21.0 was still the highest Hadoop version available [10]. In May 2012, Hadoop has moved from the 0.2x ver-
sions to the 1.0.x versions [10]. We performed a code-level comparison between versions 0.21.0 and 1.0.0. We find that the mechanisms described in this section have remained the same, thus showing that the mechanisms are not short-lived but rather are deeply rooted in Hadoop’s design philosophy. The one change we have found concerns the speculative execution algorithm. In 1.0.0, Hadoop has reverted to an older algorithm found in versions 0.20.x. Unfortunately, that particular algorithm has already been shown to have serious inefficiencies [97], a conclusion which lead to the development of the improved algorithm that we analyze in this chapter.

For brevity we say that a task was *speculated* when a new instance of the task was speculatively executed. We distinguish between the *initial* instance of a task and subsequent speculative instances of the same task. We use WTO, RTO and CTO to signify write, read and connect timeouts.

### 3.2.1 How Hadoop Deals with TaskTracker Failures

Hadoop infers TaskTracker failures by comparing per-task state variables against tunable threshold values. Table 3.1 lists the variables used by Hadoop. These variables are constantly updated by Hadoop during the course of a job. For clarity, we omit the names of the thresholds and instead use their default numerical values.

As we examine in detail the decisions related to TaskTracker failures, it shall become apparent that tolerating network congestion and compute node overload is a key driver of many aspects of Hadoop’s design. It also seems that Hadoop attributes non-responsiveness primarily to congestion or overload rather than to failure, and has no effective way of differentiating the two cases. To highlight some findings:

- Hadoop is willing to wait for non-responsive nodes for a long time (on the order of 10 minutes). This conservative design allows Hadoop to tolerate non-responsiveness caused by network congestion or compute node overload.

- A *completed* mapper task whose output data is inaccessible is re-executed very con-
servatively. This makes sense if the inaccessibility of the data is rooted in congestion or overload. This design decision is in stark contrast to the much more aggressive speculative re-execution of straggler tasks that are still running [97].

- The health of a reducer is defined as a function of the progress of the shuffle phase (i.e. the number of successfully copied mapper outputs). However, Hadoop ignores the underlying cause of unsuccessful shuffles.

### 3.2.1.1 Declaring a TaskTracker Dead

TaskTrackers send heartbeats to the JobTracker every 3s. The JobTracker detects TaskTracker failures by checking every 200s if any TaskTracker has not sent heartbeats for at least 600s. If a TaskTracker is declared dead, the tasks running on it at failure time are restarted on other nodes. Mapper tasks that completed on the dead TaskTracker are also restarted if the job is still in progress and contains any reducers.

Note that the timeout value of 600s may seem at first glance to be an arbitrarily chosen value that is high enough to likely handle most transient unavailability events. In fact this value corresponds well with insights from recent research articles that present extensive measurements of unavailability events in large clusters. A recent Google article [52] asserts that less than 10% of unavailability events lasted longer than 15 minutes. The data analyzed for this study is comprehensive and comprised tens of Google storage cells, each with 1000 to 7000 nodes, over a one year period [52]. As a result, the Google File System (GFS) [53] uses a 15 minute timeout before starting the recovery process for the data present on unavailable nodes.

### 3.2.1.2 Declaring Mapper Outputs Lost

The loss of a TaskTracker makes all mapper outputs it stores inaccessible to reducers. Hadoop re-computes a mapper output early (i.e. does not wait for the TaskTracker to be declared dead) if the JobTracker receives enough notifications that reducers are unable to
obtain the mapper output. The output of mapper M is re-computed if:

\[ N_j(M) > 0.5 \times R_j \quad \text{and} \quad N_j(M) \geq 3. \]

Let L be the list of mapper outputs that a reducer R wants to copy from TaskTracker H. A notification is sent immediately if a read error occurs while R is copying the output of some mapper M1 in L. \( F_j^R(M) \) is incremented only for M1 in this case. If on the other hand R cannot connect to H, \( F_j^R(M) \) is increased by 1 for every mapper M in L. If, after several unsuccessful connection attempts \( F_j^R(M) \mod 10 = 0 \) for some M, then the TaskTracker responsible for R sends a notification to the JobTracker that R cannot copy M’s output. A back-off mechanism is used to dictate how soon after a connection error a node can be contacted again for mapper outputs. After every connection error, for every mapper M for which \( F_j^R(M) \) is incremented, a penalty is computed for the node running M:

\[
\text{penalty} = 10 \times (1.3)^{F_j^R(M)}.
\]

A new timer is set to penalty seconds in the future. Whenever a timer fires another connection is attempted.

### 3.2.1.3 Declaring a Reducer Faulty

A TaskTracker considers a reducer running on it to be faulty if the reducer failed too many times to copy mapper outputs. Three conditions need to be simultaneously true for a reducer to be considered faulty. First,

\[
K_j^R \geq 0.5 \times A_j^R.
\]

In other words at least 50% of all shuffles attempted by reducer R need to fail. Second, either

\[
S_j^R \geq 5 \quad \text{or} \quad S_j^R = M_j - D_j^R.
\]
Third, either the reducer has not progressed enough or it has been stalled for much of its expected lifetime.

\[ D_j^R < 0.5 * M_j \quad \text{or} \quad T_j^R \geq 0.5 * \max(P_j^R, Q_j) \]

Note that for Hadoop only the existence of a connection failure is important but not the cause of the failure.

### 3.2.2 How Hadoop Deals with DataNode Failures

Hadoop detects DataNode failures using connection errors and timeouts. If a timeout expires or an existing connection is broken, a read (or write) operation is restarted with new source (or destination) DataNodes obtained from the NameNode.

The timeouts used by HDFS requests to recover from DataNode failures are conservatively chosen, likely in order to accommodate transient congestion episodes which are known to be common in data centers [31]. Both an initial task and a speculative task can suffer from these timeouts. RTOs and CTOs are on the order of 60s while the WTOs are on the order of 480s. Differences of 5s-15s in absolute timeout values exist and depend on the position of a DataNode in the HDFS write pipeline. For the argument in this chapter these minute differences are inconsequential.

### 3.3 Experiments Exposing the Inefficiencies of Failure Detection and Recovery in Hadoop

#### 3.3.1 Methodology

For our experiments we used 15 machines distributed over 4 racks in the OpenCirrus testbed [16]. One node is reserved for the JobTracker and the NameNode. The rest of the nodes run DataNode and TaskTracker processes. Each node has 2 quad-core Intel Xeon E5420 2.50 Ghz CPUs. The network is 10 to 1 oversubscribed. We run Hadoop 0.21.0 with the default configuration. Importantly, the compute nodes as well as the network were not shared with
other users. The resources were solely used by our Hadoop jobs. Even more, the compute nodes were not virtualized. As a result the performance of our testbed was predictable. This allows us to clearly identify the performance variations caused by Hadoop’s design.

We independently study DataNode and TaskTracker failures because in many Hadoop deployments DataNodes and TaskTrackers are collocated and therefore, under compute node failures it would be hard to single-out the underlying cause. We first analyze Task-Tracker failures. After the TaskTracker failure experiments one of the compute nodes became permanently disabled because of a hardware issue and this left us with one less compute node for the DataNode failure analysis. Fortunately, the two sets of results are independent, therefore this failure does not affect our findings.

The job we use in this chapter sorts 10GB of random data using 2 mapper and 2 reducer slots per node. In the experiments, we vary the number of reducers and the number of reducer waves. 200 runs are performed for each experiment. Without failures the job takes on average 220s to complete. We chose this relatively short job because current studies show the significant popularity of short jobs in cloud workloads [97, 42, 37, 36]. Our goal is not to exhaustively and quantitatively analyze Hadoop performance over many job and failure types. Instead, our aim is to expose the inefficiencies, the subtle interactions and the underlying design decisions that characterize Hadoop’s behavior under failures. Nevertheless, we argue that the thorough understanding obtained from our work is also insightful for longer jobs and for jobs running on larger deployments because Hadoop’s failure detection and recovery algorithms (subsection §3.2) remain the same. Our work is equally insightful for the case when multiple failures occur (which are more probable in larger deployments or longer jobs) because Hadoop treats these multiple failures as several single failures.

We consider TaskTracker and DataNode process failures as well as the failure of the entire compute node running these processes. The difference between the two failure types lies in the existence of TCP reset (RST) packets which are sent by the host operating system on connection attempts to the dead TaskTracker or DataNode processes. Thus, RST packets
may serve as an early failure signal. We induce the single DataNode (or TaskTracker) fail-
stop failures by randomly killing one of the DataNodes (or TaskTrackers) at a random
time after the job is started and before the 220s mark. At the end of each run we restart
Hadoop. We emulate a fail-stop failure of the entire compute node running a DataNode
(or TaskTracker) by filtering all RST packets sent after the failure if the source port of the
RST packet corresponds to the ports used by the failed DataNode (or TaskTracker). In our
experiments we look at how failures impact the job running time and the job start-up time.
We consider the job start-up time to be the time between the job submission and the job
start as there are no job queueing delays in our experiments. We consider the job running
time to be the time between the job start and job end.

Following, we present a quick roadmap of the subsequent experiments. Details and
arguments for our experimental choices can be found alongside the experiments. In the
first experiment we analyze in great detail TaskTracker process failures and find significant
performance variations. The subsequent three experiments confirm and expand on our
findings for an increased number of reducers, for running a second concurrent job and
for emulating TaskTracker node failures. For DataNode failures, we start by explaining
delayed speculative execution (an important inefficiency) using one sample run. We then
analyze this inefficiency over three experiments each with a different number of reducers or
reducer waves. We then change the speculative execution algorithm. We use the LATE [97]
speculative execution algorithm and show the downside of Hadoop’s philosophy of not
sharing failure information between its tasks. Our last experiment shows that DataNode
failures can even significantly affect the job start-up phase.

3.3.2 TaskTracker Failure Analysis

Our first experiment details Hadoop’s behavior under TaskTracker process failures. For this
first experiment we chose process failures because the presence of RST packets enabled
us to perform a more thorough analysis. In the absence of RST packets, CTOs would
slow down Hadoop’s reaction considerably and would mask the effect of important subtle
Figure 3.1: Clusters of job running times under TaskTracker failure. Without any failures the average job running time is 220s.

3.3.2.1 Detailed Analysis

Figure 3.1 plots job running time against TaskTracker failure injection time. 14 reducers in 1 wave are used in this experiment. Out of 200 runs, 193 are plotted and 7 failed. Note the large variation in job running time. This is due to a large variation in the efficiency of Hadoop’s failure detection and recovery mechanisms. To explain, we cluster the results into 8 groups based on the underlying causes. The first 7 groups are depicted in the figure. The 7 failed runs form group G8. The highlights that the reader may want to keep in mind are:

- When the failure affects few reducers, failure detection and recovery is exacerbated.
- Detection and recovery time in Hadoop is unpredictable – an undesirable property in
a distributed system. The time it takes reducers to send notifications is variable and so is the time necessary to detect TaskTracker failures.

- The mechanisms used to detect lost mapper outputs and faulty reducers interact badly. As a result, otherwise localized failures propagate in the system: many healthy reducers die unnecessarily as a result of attempting connections to a failed TaskTracker. This leads to unnecessary re-executions of reducers, thus exacerbating recovery.

**Group G1.** In G1, at least one mapper output on the failed TaskTracker was copied by all reducers before the failure. After the failure, the reducer on the failed TaskTracker is speculated on another node and it will be unable to obtain the mapper outputs located on the failed TaskTracker. According to the penalty computation (subsection §3.2.1.2) the speculated reducer needs 10 failed connection attempts to the failed TaskTracker (416s in total) before a notification about the lost mapper outputs can be sent. For this one reducer to send 3 notifications and trigger the re-computation of a mapper, more than 1200s (i.e. 3 notifications each necessitating 416s) would typically be necessary. The other reducers, even though still running, do not help send notifications because they already copied the lost mapper outputs. Thus, the TaskTracker timeout (subsection §3.2.1.1) expires first. Only then are the mappers on the failed TaskTracker restarted. This explains the large job running times in G1 and their constancy. G1 shows that the efficiency of failure detection and recovery in Hadoop is impacted when few reducers are affected and mapper outputs are lost.

**Group G2.** This group differs from G1 only in that the job running time is further increased by roughly 200s. This is caused by the mechanism Hadoop uses to check for failed TaskTrackers (subsection §3.2.1.1). To explain, let $D$ be the interval between checks, $T_f$ the time of the failure, $T_d$ the time the failure is detected, $T_c$ the time the last check would be performed if no failures occurred. Also let $n \ast D$ be the time after which a TaskTracker is declared dead for not sending any heartbeats. For G1, $T_f < T_c$ and therefore $T_d = T_c + n \ast D$. However, for G2, $T_f > T_c$ and as a result $T_d = T_c + D + n \ast D$. In
Figure 3.2: Illustration of early notifications. The tuple format is (map name, time the penalty expires, $F^R_j(M)$). For example, (A,3,2) means at time 3 a new connection should be attempted to get mapper output A, and there have been 2 failed attempts so far. The tuple values are taken immediately after the corresponding timestamp. This example considers that notifications are sent when $F^R_j(M) = 5$. Note that this occurs at different moments, shown by rectangles.

Hadoop, by default, $D = 200s$ and $n = 3$. The difference between $T_d$ for the two groups is exactly the 200s that distinguish G2 from G1. In conclusion, the timing of the TaskTracker failure with respect to the JobTracker checks can further increase job running time.

**Group G3.** In G3, the reducer on the failed TaskTracker is also speculated but sends notifications considerably earlier than the usual 416s. We call these early notifications. 3 early notifications are sent and this causes the mapper outputs to be re-computed before the TaskTracker timeout expires (subsection §3.2.1.2). To explain early notifications consider the example in Figure 3.2 where, for simplicity, the penalty (subsection §3.2.1.2) is assumed linear ($penalty = F^R_j(M)$) and the threshold for sending notifications is 5. Reducer R needs to copy the output of two mappers A and B located on the same node. Case a) shows regular notifications and occurs when connections to the node cannot be established.

Case b) shows early notifications and can be caused by a read error during the copy of A’s output. Due to the read error, only $F^R_j(A)$ is initially incremented. This desynchronization between $F^R_j(A)$ and $F^R_j(B)$ causes the connections to the node to be attempted more frequently. As a result, failure counts increase faster and notifications are sent earlier.
Because the real function for calculating penalties in Hadoop is exponential (subsection §3.2.1.2), a faster increase in the failure counts translates into large savings in time. As a result of early notifications, runs in G3 finish by as much as 300s faster than the runs in group G1.

**Group G4.** For G4, the failure occurs after the first mapper wave but before any of the mapper outputs from the first mapper wave is copied by all reducers. With multiple reducers still requiring the lost outputs, the JobTracker receives enough notifications to start the mapper output re-computation (subsection §3.2.1.2) before the TaskTracker timeout expires. The trait of the runs in G4 is that not enough early notifications are sent to trigger the re-computation of mapper outputs even earlier.

**Group G5.** As opposed to G4, in G5 enough early notifications are sent to trigger mapper output re-computation earlier.

**Group G6.** The failure occurs during the first mapper wave, so no mapper outputs are lost. The mappers on the failed TaskTracker are speculated and this overlaps with subsequent mapper waves. As a result, there is no noticeable impact on the job running time.

**Group G7.** This group contains runs where the TaskTracker was failed after all its tasks finished running correctly. As a result, the job running time is not affected.

**Group G8.** This group contains the failed jobs. The failed jobs are caused by Hadoop’s default behavior to abort a job if one of the job’s tasks fails 4 times. A reduce task can fail 4 times because of the induced death problem described next.

### 3.3.2.2 Induced Reducer Death

In several groups we encounter the problem of induced reducer death. Otherwise localized failures propagate to healthy tasks in the system. This is the case for the reducers which, although running on healthy nodes, die as a result of repeated, unsuccessful attempts to connect to the failed TaskTracker. Such a reducer dies (possibly after sending notifications) because a large percent of its shuffle attempts failed, it is stalled for too long and it copied
all mapper output but the failed ones (subsection §3.2.1.3). We also see reducers die within seconds of their start (without having sent notifications) because the conditions in (subsection §3.2.1.3) become temporarily true when the failed node is chosen among the first nodes to connect to. In this case most of the shuffles fail and there is little progress made. Induced reducer death wastes time by causing task re-execution and wastes resources since shuffles need to be repeated.

3.3.2.3 Effect of Alternative Configurations

Subsection §3.2.1 suggests failure detection is sensitive to the number of reducers. We increase the number of reducers to 56 and the number of reducer slots to 6 per node. Figure 3.3 shows the results. Considerably fewer runs rely on the expiration of the TaskTracker timeout compared to the 14 reducer case because more reducers means more chances to send enough notifications to trigger mapper output re-computation before the TaskTracker timeout expires. However, Hadoop still behaves unpredictably. The variation in job running time is more pronounced for 56 reducers because each reducer can behave differently: it
can suffer from induced death or send notifications early. With a larger number of reducers these different behaviors can combine and lead to many different outcomes.

Next, we run two concurrent instances of the 14 reducer job and analyze the effect that
the second scheduled job has on the running time of the first. A failure is injected the same way as described before. Figure 3.4 shows the results for the first scheduled job compared to the case when it runs alone. Without failures, the first scheduled job finishes after a baseline time of roughly 400s. The increase from 220s to 400s is caused by the contention with the second job. The large variation in running times is still present under failure. The second job does not directly help detect the failure faster because the counters detailed in subsection §3.2.1 are defined per job. However, the presence of the second job indirectly influences the first job. Contention causes longer task running times and in Hadoop this leads to increased speculative execution of reducers. A larger percentage of jobs finish around the baseline time because sometimes the reducer on the failed TaskTracker is speculated before the failure and copies the mapper outputs that will become lost. This increased speculative execution also leads to more notifications so fewer jobs rely on the TaskTracker timeout expiration. Note also the running times around 850s. These jobs rely on the TaskTracker timeout expiration but also suffer from the contention with the second job.

The next experiment emulates the failure of a whole node running a TaskTracker process. Results are shown in Figure 3.5 for the 56 reducer job. The lack of RST packets means that every connection attempt is subject to a 180s timeout. Therefore, there is not enough time for the reducers to send notifications, so all jobs impacted by the failure rely on the TaskTracker timeout expiration in order to continue. Moreover, reducers finish only after all their pending connections finish. If a pending connection is stuck waiting for the 180s timeout to expire, this stalls the whole reducer. This delay can also cause speculative execution and therefore increased network contention. These factors are responsible for the variation in running time starting with 850s.

3.3.3 DataNode Failure Analysis - Delayed Speculative Execution

The DataNode failure experiments emulate the failure of a whole compute node running a DataNode process. Thus, RST packets do not appear. We do not present the effect of DataNode process failures since their impact is low. While a DataNode failure is expected
to cause some job running time variation and performance degradation, the speculative execution algorithm should eliminate significant negative effects. Our results show the opposite. As a quick example consider Figure 3.8. In this experiment, the speculative...
execution algorithm is largely ineffective after the map phase finishes (80s). The complete results for different number of reducers and reducer waves are plotted in Figures 3.8, 3.9, and 3.10. Figure 3.11 shows the CDF of job running times for the three above-mentioned
Figure 3.10: CDF of job running times for the 52 reducers, 4 waves case

Figure 3.11: CDF of job running times for Figures 3.8, 3.9 and 3.10.

experiments.
3.3.3.1 Understanding Delayed Speculative Execution

To understand the DataNode failure results, we first take a deeper look at the interactions between DataNode failures and the speculative execution algorithm. We show that these interactions can cause a detrimental effect which we deem delayed speculative execution. This consists in one speculative execution substantially delaying future speculative executions, or in the extreme case precluding any future speculative executions. The reason lies with the statistical nature of Hadoop’s speculative execution algorithm (subsection §2.2.3).

To explain delayed speculative execution, consider the sample run in Figure 3.6 which plots the progress rates of two reducers alongside the value of the left side of equation (2.1) from subsection §2.2.3. We call this left side the limit. For this run, 13 reducers are started in total, all in 1 wave. In Hadoop, the progress rate for one task is the maximum rate over the progress rates of the initial task instance and the speculative instance. When an instance of the task completes, the final progress rate for the task is that of the completed instance. Initially, the reducers need to wait for the map phase to end. Their progress rates are close to the average rate and the standard deviation is small. Hence, the limit is relatively high and close to the average rate. A DataNode failure occurs at time 176s and affects reducers R9 and R11. The failure interrupted the write phase of these reducers and therefore R9 and R11 are stuck in a WTO. At time 200s, R9 is speculated. The progress rate for the speculated R9 is very high because it does not need to wait for mapper outputs to be computed. The outputs are readily available for copying. The sort phase is also fast and this helps further increase the progress rate of the speculative R9. In Figure 3.6 this high rate is visible as a sudden large spike. As a result of the first spike, the average rate increases but the standard deviation increases more. Consequently, the limit decreases. Around 200s, the progress rate for R9 decreases because the speculative R9 needs time to finish the write phase. Because of this progress rate decrease the limit increases but not to the point where it would allow R11 to be speculated. R11 is speculated only around 450s when its progress rate becomes lower than the limit due to the prolong stall in the WTO. In the extreme case, if the limit is lowered too much (can even become negative)
then no further speculative execution may be possible. To continue, all reducers stuck in an WTO would need to wait for the WTO to expire, because they cannot be speculated. In the general case, spikes need not be isolated as in our example. Several reducers can have progress rate spikes at the same time.

The influence that a speculative execution has on the limit and consequently the delay it creates for subsequent speculative executions depends on the shape of the spike it creates. We plot these shapes in Figure 3.7. The ascending part of the spike decreases the limit. The severity of this ascending part depends on the amount of data the reducer needs to shuffle and on the speed of the network transfers. If little data is necessary or network transfers are very fast, then the reducer quickly finishes the shuffle and sort phases with a very high progress rate. The decreasing part of the spike influences how much the limit increases. In our runs we see three distinct decreasing shapes each of which influences the limit differently. A short decrease signals that the write phase proceeded normally (reducer R10). A longer decrease signals that the speculative task also encountered a CTO because of the DataNode failure (reducer R12). A sharp decrease signals that the initial reducer finished shortly after the speculative reducer finished the shuffle and sort phases (reducer R2).

3.3.3.2 Effects of Delayed Speculative Execution on the Reduce Phase

Next, we explain in detail the results for DataNode failures injected during the reduce phase but after the map phase ends at roughly 80s.

For the 52-reducer, 1-wave case in Figure 3.8 the Hadoop speculative execution algorithm is ineffective after the map phase ends (~80s). Notice the two parallel clusters of increasing job running time greater than 600s. The large job running times are caused by delayed speculative execution. Due to delayed speculative execution there is usually at least one reducer that cannot be speculated and therefore has to wait for the WTO to expire before continuing. The reason why two clusters exist lies in a Hadoop code-level design choice where a reducer does not remember a failed DataNode if it caused a WTO. Thus,
the same failed DataNode can cause the reducer to get stuck in a CTO after the WTO. On the other hand, after a CTO, the reducer remembers the failed DataNode and no further CTOs are caused by that failure. If the WTO occurs at the last block that the reducer needs to write, no CTOs can follow. Therefore, one cluster is formed by reducers suffering only from a WTO while the other cluster is comprised of reducers suffering from both a WTO and a CTO. The steady increase in job running time for each of the clusters is a function of how close to the end of the job the failure was injected.

For the 13-reducer, 1-wave case in Figure 3.9 the speculative execution algorithm is more effective after the map phase end (∼80s). Large job running times caused by delayed speculative execution are still common but faster running times also exist. Compared to the 52-reducer case, each of the 13 reducers is responsible for writing 4 times more blocks and this considerably increases the chance that a CTO affects the write phase of a speculative reducer. As a result of these CTOs, the limit is increased more and speculative execution becomes possible again thus resulting in some faster running times. Moreover, with only 13 reducers, less speculative executions are necessary overall since fewer reducers are impacted by the failure. Sometimes only 1 or 2 speculative executions are necessary overall and if both are started at the same time (in the first spike) there is no other speculative execution to be delayed.

For the 52-reducer, 4-wave case presented in Figure 3.10 the speculative execution algorithm performs well. The reason is that the reducers in the last 3 waves all have very big progress rates since the mapper outputs are already available and the sort phase is fast. As a result, the limit increases and is less influenced by subsequent spikes. Consequently, further speculative execution is not impaired.

3.3.3.3 Effects of Speculative Execution on the Map Phase

We next explain the results for the experiments in Figures 3.8, 3.9 and 3.10 when the failure is injected before 80s. During the first 80s, failures overlap with the map phase and reducers are not yet in the write phase. We did not see cases of delayed speculative execution for
the map phase because mappers, unlike reducers, did not have to wait for their input data to be available and the progress rates of different mappers were similar. In theory, delayed speculative execution is also possible for the map phase when there is a large variation in progress rates among mappers. This can happen when using a network topology with variable bandwidth. In these cases fast progressing mappers could skew the statistics.

Nevertheless, for the map phase we also identified speculative execution inefficiencies under DataNode failures. We encounter needless speculative execution caused by not including in the decision process information about why a task is slow. For example, a mapper task can stall on a 60s CTO but the speculative execution algorithm speculates a task only after the task has run for at least 60s. The speculative execution can be needless here because it oftentimes occurs exactly when the CTO expires and the initial mapper task can continue and quickly finish.

When the failure occurs during the map phase, the job running times are smaller than when reducers are affected. However, job running time variation still exists and is caused by several factors most of which are common to all 3 experiments from Figures 3.8, 3.9 and 3.10. For example, sometimes the NameNode encounters a CTO at the end of a job, when it writes to HDFS a file with details about the run. This delays the delivery of the job results to the user even though all tasks, and therefore the computation are finished. Also, if one of the mappers from the last mapper wave suffers from a CTO this impacts job running time more since the CTO cannot be overlapped with other mapper waves. The reducers have to wait until the mapper stuck in the CTO finishes. Specific to the 52-reducer, 4-wave case (Figure 3.10) is the fact that timeouts are possibly encountered by reducers in every of the 4 waves. As a result, job running times are slightly larger for this scenario.

### 3.3.4 DataNode Failure Analysis - Not Sharing Failure Information

In this section, we show the effect of Hadoop’s philosophy to trade-off sharing of potentially useful information for extreme scalability. The effect is a significant increase in job-running time caused by failures being re-discovered by each task separately.
3.3.4.1 Using LATE as an Alternative Speculative Execution Algorithm

We chose LATE as the speculative execution algorithm in this experiment because its goal is to react to under-performing tasks as early as possible. We first look at the 52-reducer,
1-wave case. As suggested in [97] we set the LATE SlowTaskThreshold to the 25th percentile. The results are plotted in Figures 3.12 and 3.13.

Overall, LATE performs better than Hadoop’s speculative execution algorithm but running times larger than 600s are still present. Because of its more aggressive nature, LATE oftentimes speculates a task before the failure and therefore tasks having both the initial and the speculative instance running before the failure are present. The large job running times in this experiment are the runs in which both the initial task instance and its speculative instance are stuck in a WTO because they are affected by the same DataNode failure. The task can continue and finish only after the WTO expires for one of its instances. Note that Hadoop does not implement sharing of failure information between tasks and therefore the failure is re-discovered individually by each task.

For the 52-reducer 4-wave and the 13-reducer 1-wave cases LATE did not produce large job running times. In these cases, in our experiments the problem described above is still possible but it is less probable since fewer reducers are active at the same time. Note that it is enough for just one task to be affected in the manner described and the job running time is significantly affected.
3.3.4.2 Delayed Job Start-up

We now analyze the effect of DataNode failures on the job start-up time. On each run, we fail one random DataNode at a random time starting 5s before the job submission time and ending 5s after. Job submission time is at 5s. The results are pictured in Figures 3.14 and 3.15. Without DataNode failures, the job start-up time is roughly 1s, thus the job starts soon after second 6. This explains why failures occurring after 6s in Figure 3.14 do not impact job start-up time. However, if the DataNode failure occurs during the first 6s most jobs are impacted.

The reason is that even before a job is started, the JobTracker needs to make multiple HDFS write requests to replicate job-specific files. By default, 6 such files are written: job.jar (java classes for the job), job.split and job.splitmetainfo (description of map inputs), job.xml (job parameter values), jobToken (security permissions) and job.info. Any timeout delaying the writing of these files delays the start of the whole job. However, Hadoop does not share failure information between these 6 write operations so several of these writes could suffer from timeouts. Moreover, the job.jar file is replicated by default using a large replication factor of 10 [93]. This makes this operation even more susceptible to DataNode failures. Unfortunately, this large replication factor is not adaptive and can cause inefficiencies when failures occur in small clusters. In our runs, with 13 total DataNodes and 10 DataNodes required by the large replication factor for job.jar, the chance that the write operation was impacted by the randomly induced failure was big. This explains why only few runs in Figure 3.14 were unaffected by a failure injected before 6s.

3.3.4.3 The Effect of Not Sharing Failure Information on the Map Phase

Having understood that Hadoop does not share failure information between tasks we can also apply this to the experiments in Figures 3.8, 3.9 and 3.10, for the case when the failure is injected before 80s. Several mappers are influenced by the failure. This is because each mapper task performs 3 or 5 HDFS block read operations for processing one single input split. The first access is for the job.split file which identifies the input split for the job. The
second access reads the input data while the third access reads the start of the subsequent block because a mapper input split can span HDFS block boundaries. Two more accesses can appear in case the mapper input split is not at the beginning of the HDFS file. Generally, the more HDFS accesses a task performs the greater the chance a failure will impact the task. Since Hadoop does not share failure information between mappers, many mappers can encounter a CTO because of the same failure.

3.4 Discussions and Implications

Delayed speculative execution is a general problem. Delayed speculative execution is a general concern for statistical speculative execution algorithms such as Hadoop’s. There are many ways to trigger delayed speculative execution and failures are just one of them. The large HDFS timeouts are not a fundamental cause of delayed speculative execution, although they can add to the overhead. Two common conditions are needed to trigger the negative effects of delayed speculative execution: the existence of slow tasks that would benefit from speculative execution and conditions for tasks to suddenly speed up and lead
to progress rate spikes. Slow tasks have many common causes including failures, timeouts, slow machines or slow network transfers. Progress rate spikes can be caused by varying input data availability (the first reducer wave needs to wait for the map phase to finish but subsequent reducer waves need not wait) or by small reducer input data size (small input size means fast progress). Varying network speeds can also cause progress rate spikes. This especially concerns recent proposals for circuit-augmented network topologies [90, 49] that inherently present large variations in bandwidth over different paths.

With performance variation becoming a fact of life in the cloud it would be useful to develop speculative execution algorithms that forgo statistics altogether and instead are centered around a thorough understanding of the computation performed and on the current performance of the infrastructure. For example, static analysis of jobs [55, 60] could be used to generate a performance model that can be subsequently leveraged for scheduling and speculative execution decisions.

**Not sharing failure information.** We have shown the negative effects of not sharing failure information at job runtime. Regardless of how good a speculative execution decision may be before a failure, all benefits can quickly be invalidated when a speculated task is affected by a failure because it needs to read or write data to the failed node. Therefore, good speculative execution decisions are not sufficient. Care must be taken at runtime to ensure the success of the speculative execution decision. Sharing failure information at runtime is one potential approach to ensure this success. The benefits of sharing failure information are not limited to speculated tasks. Different tasks can significantly benefit from sharing failure information because they would not have to individually re-discover a failure.

We believe that sharing need not be limited to failure information. Performance, scalability or straggler information could also potentially be shared not only inside one application but among similar cloud applications [45]. With more information obtained from sharing, a large scale computing framework such as Hadoop would be more likely to make better provisioning and runtime decisions. Nevertheless, Hadoop’s reason for not sharing
information is warranted. Given the unprecedented scale of today’s cloud environment sharing information between tasks, if not carefully done, can quickly become a serious bottleneck. Moreover, the extreme scalability of Hadoop’s design is a cornerstone of its success [85, 87]. Future work can analyze what is the minimum amount of information that, if shared, can yield important benefits. Also, it is useful to analyze the trade-off between the shared information’s freshness and the overall gain in performance.

**Decoupling failure recovery from overload recovery.** TCP connection failures are not only an indication of task failures but also of congestion. However, the two factors require different actions. It makes sense to restart a reducer placed disadvantageously in a network position susceptible to recurring congestion. However, it is inefficient to restart a reducer because it cannot connect to a failed TaskTracker. The data will still be unavailable regardless of whether the reducer is restarted or not. Unfortunately, the news of a connection failure does not by itself help Hadoop distinguish the underlying cause. This overloading of connection failure semantics ultimately leads to a more fragile system as exemplified by the induced reducer death problem.

In the future, it can prove useful to decouple failure recovery from overload recovery entirely. To do this, Hadoop needs to consider and obtain more environmental information. For example, for dealing with compute node overload, solutions can leverage the history of a compute node’s behavior which has been shown to be a good predictor of transient compute node overload over short time-scales [27]. Nevertheless, currently Hadoop seems to function primarily at the application-layer, managing the computation and detecting unexpected environmental conditions using simple and oftentimes simplistic approaches. More intrusive approaches can provide significantly more information at the cost of trading-off some of the simplicity. The challenge is not avoiding this path altogether but rather understanding how much to trade-off. One example of a slightly more intrusive approach is using transport protocol information for inferring network congestion. One approach is to consider the use of network protocols such as AQM/ECN [51, 76, 44] that expose more information to the applications. Another approach is to extend the functionality of Hadoop
to actively monitor the infrastructure. One example of a system that performs this function is the Falcon spy network [64]. This system provides fast failure detection times by coordinating a network of spies, each monitoring one layer of the system using platform-specific logic [64].

The need for adaptivity. Timeouts and thresholds in Hadoop are static. The disadvantage of static timeouts is that they cannot correctly handle all situations. Conservative timeouts are useful to cause a task’s progress rate to decrease in order to be noticed by the speculative execution algorithm. Conservative timeouts are also useful for protection against temporary network or compute node overload. However, short timeouts may allow fail-over to other DataNodes for read and write operations. TaskTracker thresholds are also static and we have shown that this leads to poor performance when few reducers are impacted by a TaskTracker failure.

Future work should consider adaptive timeouts that are set using system wide information about congestion, state of a job and availability of data. As a more general solution it would prove useful to complement Hadoop’s design with a dependable failure detection and performance measuring mechanism, that would go beyond timeouts and guesswork - approaches that we have shown to be inadequate today. Hadoop needs to be much more aware of its environment and adapt to performance-influencing environmental characteristics: reliability, sharing of resources, use of virtualization, performance variability.

The need for analysis work on large scale computing frameworks. Our work is the first to provide a thorough analysis of Hadoop’s performance under failure conditions. We believe such analysis work is fundamental for improving application performance in cloud environments. There is already a large body of work analyzing the performance of representative cloud infrastructures [80, 58, 92, 54]. We think this should be complemented with analysis work on representative cloud applications especially given their large but still increasing popularity. We hope our work is an insightful first step to this end.

We have shown that Hadoop’s internal mechanisms cause significant and unpredictable performance variations under failures. These results suggest that it is challenging to model
Hadoop’s performance under failure conditions. Comparing our results to recent work on simulating the performance of Hadoop [91] under failures highlights the difficulty in developing accurate models of Hadoop’s behavior based mainly on Hadoop’s high-level design specifications. The subtle interactions that lead to performance variations do not appear in the model. Nevertheless such modeling work is very important in the cloud since users need to be able to estimate application performance in order to choose suitable large scale computing frameworks or cloud environments. We believe analysis work such as ours can be leveraged in the development of more advanced models of Hadoop’s behavior.

3.5 Related Work

Prior work has tackled the problem of surviving compute node failures. For example, smart replication of intra-job intermediate data (e.g. mapper outputs) has been proposed to improve performance under TaskTracker failures [63, 27]. Replication minimizes the need for re-computation of intermediate data and allows for fast failover if one replica cannot be contacted as a result of a failure. Unfortunately, replication may not be always beneficial. It has been shown [63] that replicating intermediate data guards against certain failures at the cost of overhead during periods without failures. Moreover, replication can aggravate the severity of existing hot-spots. Therefore, complementing replication with an understanding of failure detection and recovery is equally important. Existing work on leveraging opportunistic environments for large distributed computations [66] can also benefit from this understanding as such environments exhibit behavior that is similar to failures.

A recent study [54] characterizes the effect of network-related failures in the cloud. While the study does not deal with application-level effects, it shows network-related failures have an important impact on data transfers. Applications built on top of large scale computing frameworks like Hadoop typically rely heavily on data transfers. Our study paints a complementary picture to the impact of network-related failures. We analyze compute node failures and specifically target application-level design inefficiencies and inter-
actions.

Other recent studies have found and analyzed significant cloud performance variability [80, 58, 92, 82]. The variability detected by these studies mainly stems from environmental causes such as sharing the data center network or using virtualized environments. Our work complements these studies by identifying and analyzing the significant performance variation caused by the design of a cloud application. We showed that this variation can appear even in cloud environments with predictable performance.

Our work is also related to recent efforts for improving the performance of speculative execution algorithms in large scale computing frameworks [97, 27]. There are however important differences. Our work goes beyond speculative execution. This related body of work does not consider failure detection but we do so in detail. These related studies only relate to failure indirectly through outliers which are one possible effect of failures. Instead we analyze the effect of failures directly and exhaustively. We find that failures interact with Hadoop’s inner workings in subtle ways (e.g. induced reducer death) and are at odds with Hadoop’s design decisions (not sharing any information for scalability reasons). We have even discovered possible improvements to this past body of work. We analyzed the LATE algorithm [97] and showed it can be improved under failures. Moreover, the delayed speculative execution problem we uncovered is a new and important concern for statistical speculative execution algorithms.

3.6 Chapter Summary

In this chapter we exposed and analyzed Hadoop’s sluggish, variable and unpredictable performance under compute node failures. We identified the mechanisms and design decisions responsible and pointed out the limitations that ultimately resulted in the inefficient behavior. We believe our findings are generally insightful beyond Hadoop and will pave the way for a new class of more advanced large scale computing frameworks that are more predictable and more robust.
Chapter 4

**RCMP: A System Enabling Efficient Re-computation Based Failure Resilience for Big Data Analytics**

Data replication, the main failure resilience strategy used for big data analytics jobs, can be unnecessarily inefficient. It can cause serious performance degradation when applied to intermediate job outputs in multi-job computations. For instance, for I/O-intensive big data jobs, data replication is especially expensive because very large datasets need to be replicated. Reducing the number of replicas is not a satisfactory solution as it only aggravates a fundamental limitation of data replication: its failure resilience guarantees are limited by the number of available replicas. When all replicas of some piece of intermediate job output are lost, cascading job re-computations may be required for recovery.

In this chapter we show how job re-computation can be made a first-order failure resilience strategy for big data analytics. The need for data replication can thus be significantly reduced. We present RCMP, a system that performs efficient job re-computation. RCMP can persist task outputs across jobs and leverage them to minimize the work performed during job re-computations. More importantly, RCMP addresses two important challenges that appear during job re-computations. The first is efficiently utilizing the available compute node parallelism. The second is dealing with hot-spots. RCMP handles both by switching to a finer-grained task scheduling granularity for re-computations. Compared to RCMP, in our experiments, data replication is 30% to 100% worse during failure-free periods. More importantly, RCMP yields comparable or even better total multi-job running time even under single and double data loss events.
4.1 Motivation and Outline of RCMP’s Benefits

Data replication is the main failure resilience strategy used for big data analytics jobs today. It consists of writing several replicas (copies) of the same piece of data in different locations in the hope that under failures at least one replica will survive. Unfortunately, when applied to intermediate job outputs in multi-job computations (series of jobs with the output of one being the input of another), data replication can be greatly inefficient. This is important because multi-job computations are very popular. The primitives provided by big data processing systems (e.g. Hadoop [9], MapReduce [42]) constrain the amount of work possible in a job. As a result, users need to divide their algorithms into multiple jobs [65, 93, 61] or rely on higher level languages (e.g. Hive [86, 87] or Pig [70]) which usually also get compiled into sequences of jobs. Moreover, the division in multiple jobs facilitates the use of simple, general, computation-agnostic checkpointing mechanisms based solely on handling the outputs of intermediate jobs [39]. We are aware of one computation requiring as many as 150 jobs to complete [6].

Even writing relatively few replicas (3 replicas is common today [53, 83]) can be an expensive operation in the context of big data analytics because the large data transfers required put significant stress on the network and the storage. Today’s clusters are especially inefficient at handling large transfers due to economical constraints and architectural bottlenecks (e.g. oversubscribed networks [30], poor disk throughput [78, 81]). For instance, in our experiments we show that in the absence of failures, an I/O-intensive multi-job computation can almost double its running time when the replication factor is increased from 1 to 3. The same multi-job computation took 40% more time to complete when a replication factor of 2 was used instead of 1. Importantly, the large performance penalty induced by data replication is paid on every use of replication, even during failure-free periods.

However, reducing the number of replicas is not a satisfactory approach. This simply aggravates the inherent limitation of data replication: its failure resilience guarantees are fundamentally limited by the number of replicas of the data. Having insufficient replicas leaves computations exposed to failures and this can result in significant detrimental im-
pact on the performance of multi-job computations. The reason is that without the use of data replication, failures can easily cause data loss which can trigger cascading job re-computations: several jobs will need to be re-computed for the lost data to be re-generated. In the worst case, the re-computation may have to revert all the way to the beginning of the multi-job computation. This suggests the need for devising efficient approaches to job re-computation.

Unfortunately, efficient re-computation support is noticeably absent in today’s big data processing systems. The jobs affected by failures can be re-submitted but the system treats the re-submissions identically to the initial runs: it computes the jobs entirely. In this chapter we show that efficient job re-computation can be made a first-order failure resilience strategy for big data analytics. If done right, re-computation can be very efficient when failures do occur while bearing no cost during failure-free periods. Thus, the need for data replication can be greatly reduced. We present RCMP (RCMP is a name derived from the word re-computation) a system that performs efficient job re-computations in the context of the popular MapReduce paradigm. While extending the MapReduce model with support for efficient job re-computations is important and practically relevant given the popularity of MapReduce, we believe that our work on the importance and challenges of job re-computation transcends the MapReduce paradigm. In fact, our work should apply to any big data parallel processing computation model based on DAGs of tasks. We view re-computation not as a replacement for replication but rather as a complement. Our position is that enabling efficient re-computation will in turn enable judicious use of replication thus facilitating improvements in overall computation performance. For example, future work can analyze the use of data replication as a means of bounding the cost of re-computation.

RCMP is efficient. It re-computes only the minimum number of tasks necessary for each re-computed job. For this, RCMP persists across jobs mapper outputs as well as reducer outputs that are part of successfully completed intermediate jobs. On failures that cause data loss, RCMP decides which jobs must be re-computed and based on the persisted data it also determines the minimum number of tasks that need to be re-computed for each
Figure 4.1: The set of tasks and data transfers (both in bold) that are part of the re-computation of a MapReduce job under RCMP. M = mapper task, R = reducer task. A failure occurs just before Job 2 completes. The outputs of tasks $M_{11}$, $R_{11}$, $M_{21}$, $R_{21}$ are lost due to the failure and need to be re-computed. In contrast, existing systems re-compute everything.

re-computed job. As an example, consider Figure 4.1 which illustrates a re-computation performed by RCMP. The failure occurs just before Job 2 finishes. $R_{21}$ is lost and needs to be re-computed. But $R_{21}$ requires the output of $M_{21}$ which is also lost. In turn, $M_{21}$ is based on the output of $R_{11}$ which was also on the failed node and was lost. Thus, Job 2 cannot continue before $R_{11}$ is re-generated. Therefore, RCMP has to cascade back to Job 1 to re-generate the input of Job 2. In total, RCMP re-computes only the tasks that had outputs on the failed compute node ($M_{11}$, $R_{11}$, $M_{21}$, $R_{21}$) as well as the data transfers that are required for these re-computed tasks (bold lines in Figure 4.1). Note that the re-computation work performed by RCMP is a fraction of the work performed by current systems which re-compute everything. The objective of RCMP to maximize data reuse is shared by previous work in programming languages [75, 67] or cloud computing (Nectar [57], RDD [96]). However, determining what to re-compute is only a small part of achieving efficient job re-computations. RCMP goes beyond the what. Its uniqueness stems from improving how a job is re-computed.

We identified and tackled two fundamental challenges that limit the efficiency of job re-computations: the difficulty in fully leveraging the available compute node parallelism
and the presence of hot-spots. The first challenge is that during job re-computation, the re-computed tasks are unlikely to be numerous enough to efficiently utilize the available compute node parallelism. In other words, the task scheduling granularity used during the initial run is insufficient for efficient job re-computation. This results in underutilized compute nodes and consequently inefficient re-computation. The second challenge is that hot-spots appear during the re-computation of a job’s map phase. In the initial run of the job, mapper accesses to input are essentially balanced over all nodes and on each node they are partly serialized because not all mappers can run at once. We find that during re-computation, mapper accesses can concurrently concentrate on one or a few storage locations. The resulting contention significantly increases mapper running time and consequently the whole job re-computation time. RCMP’s approach to these challenges is to switch to a more fine-grained task scheduling granularity during re-computation. This better utilizes the available compute nodes and mitigates hot-spots by distributing computation and data accesses over all the nodes used for re-computation.

In our evaluation, we quantitatively describe the magnitude of the overheads that data replication can introduce as well as the benefits of efficient re-computation. For this we use an implementation of RCMP on top of Hadoop. In our experiments, compared to RCMP, data replication is 30% to 100% worse during failure-free periods. More importantly, RCMP yields comparable or even better total multi-job running time even under single and double data loss events. On a 11-node cluster, RCMP re-computed a single job up to 5x faster compared to a full re-computation of the same job.

In this chapter, we refer to the first execution of a job as the initial run of that job. During failure recovery, parts of the job may have to be re-executed. We call such a re-execution a re-computation run of that job.

The chapter is organized as follows. Section §4.2 discusses at length the problems and overheads of replication. Section §4.3 presents the capabilities and design details of RCMP while section §4.4 discussed additional considerations. Section §4.5 presents the evaluation of RCMP. Section §4.6 discusses related work and finally section §4.7 presents
a short summary and concludes the chapter.

4.2 Why Replication is Problematic

In this section we provide detailed arguments in support of our claim that replication is too costly to be the only failure resilience strategy used in big data analytics. Despite the failure resilience guarantees and the performance benefits that replication can offer in a few narrow cases, there are simply too many practically relevant situations in which replication costs far outweigh the benefits. This suggests the need for devising more efficient failure resilience strategies.

Part of the overhead of replication stems from inefficiencies in the design and implementation of current systems. For example, in their most common and affordable design, data center networks are often oversubscribed [30]. Moreover, the disk throughput obtained by applications frequently falls well short of the full capabilities that the disk hardware can deliver [81, 77]. A number of solutions have been proposed to improve I/O performance. These solutions could also decrease the absolute cost of replication. Some proposals advocate batching optimizations designed to mitigate the detrimental effect of excessive seeks caused by concurrent disk accesses [81, 77]. Others note that a major culprit is the layering of the distributed file systems on top of general purpose file systems which are not optimized for big data workloads [81]. Solutions leveraging raw access to disk are a promising alternative [68]. While these solutions are able to incrementally improve the performance, the fundamental limitation remains. Replication adds extra I/O workload into the system. Therefore, the relative overhead of replication is expected to persist despite the optimizations.

4.2.1 Overrated Benefits of Replication

Failures are not an ubiquitous threat Replication does provide useful failure resilience guarantees. Current replication strategies [53, 9] are able to protect against the simultaneous failure of two nodes or against single rack-level failures. This is particularly useful
when a job has a high probability of encountering a failure. One example are large scale, long-running jobs spanning thousands of nodes.

However, most data analytics users do not routinely run such large scale jobs. Only a handful of companies have extremely large clusters. Cloudera reports that the median size of a data analytics cluster is less than 30 nodes [18], while the average is around 200. At this moderate scale node failures are expected only at an interval of days [77].

Figure 4.2 depicts the rate at which compute nodes become unavailable for the STIC [20] and SUG@R [21] clusters at Rice University. STIC has 218 nodes and SUG@R has 121. The traces capture the period Sep 2009 - Sep 2012 for STIC and Jan 2009 - Sep 2012 for SUG@R. The statistics are based on the output of an automated script which checks the status of all compute nodes and reports unavailable nodes once per day. Less than 5% of the days had no reports and therefore we discarded them. Note that only 12% of days for SUG@R and 17% of days for STIC show new failures. Discussions with the IT staff revealed that most failure events reported in Figure 4.2 are likely hardware issues that take at least a day to solve. The few days with many nodes becoming unavailable are unplanned situations (scheduler and file system outages or performance degradation). Our numbers corroborate with estimates from other studies [77] and suggest that for the popular moderate-sized clusters occasional failures should be expected but are not an ubiquitous threat. Therefore, in these situations continuous use of replication for failure resilience is unwarranted.

**Data locality is oftentimes inconsequential**

If a node storing a piece of data also processes it, then the computation is said to be data local. Increased data locality can lead to improved job running times when it is significantly more efficient to process data locally. One example is a cluster with a highly oversubscribed network. In clusters where computation and storage are collocated, replication can improve the chance of scheduling data local tasks. More replicas result in better chances that a node having a replica of a task’s input data will be selected by the scheduler to run the task.

However, there are many situations in which data locality is either not applicable, is
inconsequential or easily obtainable without replication. First, note that data locality is not even applicable [68] to non-collocated environments. All transfers are remote in this case. Second, data locality is inconsequential for the collocated case when the network is not the bottleneck. Such systems are often proposed today even for the large scale [56, 24, 69] and have long been economically viable at moderate scale. Future trends point to advances in networking technologies that will outpace corresponding advancements in disk drive technologies thus eventually making disk locality completely irrelevant [25]. Third, oftentimes data locality is easily achievable without replication. Consider again the collocated case. Data locality is trivially obtained today by distributing data evenly across exactly the same set of nodes that perform computations. Thus, each node will have plenty of local data to compute on and little or no remote access is required. Moreover, improving data locality can also be done with smart scheduling decisions [95]. Even if none of the above situations apply, the benefits of data locality may not necessarily offset the overhead of replication.

**Speculative execution** Replication may also improve the chance of performing suc-
cessful speculative execution of mappers. The idea is that if a slow mapper needs to be duplicated (or restarted), the duplicate can read the input from another location thus potentially bypassing the problem that caused the initial task to be slow.

Note that this improvement applies only to the case when the slowness is caused by inefficiencies in reading input data (bad drives, slow network transfers, overloaded compute nodes). If the cause of the slowness strictly affects the computation, then speculative execution will succeed even in a single-replicated system. Moreover, the overall benefits of speculative execution should not be overestimated. Studies show that as many as 90% of speculatively executed tasks do not help improve computation time [34]. The reason is that it is hard for speculative execution algorithms to collect enough information to understand the causes of stragglers and they end up making suboptimal decisions as a result [27].

4.2.2 Indirect Costs of Replication

Increasing job running time is an obvious danger of using replication. However, replication also has several less obvious disadvantages. First, replication in one job indirectly affects other concurrently running jobs by increasing the contention on disk and network resources. Second, replication increases the costs necessary for provisioning a cluster that can sustain a given job execution rate because extra compute nodes or disks are necessary to compensate for the overhead of replication. Third, replication makes a system that collocates storage and computation harder to scale. Future projections show that the number of cores in a commodity compute node will increase significantly but this trend will not be matched by a similar increase in the throughput of commodity disk drives [25]. The only way to increase local I/O throughput will be increasing the number of disk drives. This is already challenging today as system designers are pushing the limits of solutions for cooling and chassis design by fitting as many as 24 disks per compute node [68]. The extra overhead of replication further aggravates this unsustainable trend.
4.3 Improving Re-computations with RCMP

In this section we detail the design and capabilities of RCMP. To provide a theoretical quantification of the magnitude of the challenges and of RCMP’s benefits this section uses a simple model of the environment and of the MapReduce paradigm. We make the following notations and assumptions. There are N compute nodes and each node writes to D separate storage locations (local disks or remote storage nodes). Each node is provisioned with S mapper slots and S reducer slots. Each node runs $W_M$ waves of mappers and $W_R$ waves of reducers. For simplicity consider that each compute node runs the same number of tasks and each task performs the same amount of work. We make these assumptions for illustration purposes only. RCMP does not need them. We further assume that there exists a method of tracing back key-value pairs lost on failure to the reducer that created them. One simple approach used today to achieve this is to divide the job output file into separate partitions with one partition for each reducer [83].

4.3.1 Overall System Design

We now present the design of RCMP using Figure 4.3 as the illustration. RCMP extends Hadoop’s design with advanced functionality necessary for efficient re-computation.

The initial job submission is similar to Hadoop but we describe it here for completeness. The user submits the multi-job computation and describes the job dependencies. A middleware program uses the dependencies to decide the order of job submission. A job is submitted only after the jobs that it depends upon are successfully computed. The jobs are submitted to the Master one by one. The Master possesses no knowledge of job dependencies and knows only how to run individual jobs to completion. Upon receiving a regular job (not re-computation), a job initialization component (JobInit) in the Master creates the tasks (circles in Figure 4.3) that need to be executed and the scheduler assigns them to cluster nodes.

Job execution is modified in RCMP. RCMP recognizes that during the computation of a job a significant amount of data (mapper outputs as well as reducer outputs) needs to be
materialized anyway for the job to complete. RCMP persists this data across jobs to benefit potential future re-computation, effectively trading-off storage space for re-computation speed-up. The rationale is that in the common case, failures are likely to lead to the loss of only a small portion of a job’s persisted data. Therefore, most of the data persisted on an initial run can be reused to minimize the work that would need to be performed in the event of re-computation.

Failure handling is much more advanced in RCMP. The Master periodically checks the cluster health and detects failures. If those failures cause irreversible data loss (all replicas of some data are lost), then the Master informs the middleware which files (job outputs) were affected also which specific reducer outputs were affected. The middleware then immediately cancels the currently running job since it cannot complete without the lost data. The middleware uses the job dependency information and the affected files to infer which jobs need to be re-computed and in which order so that the lost data is re-generated.
When submitting a re-computation job, the middleware tags it with the reducer outputs that need to be re-computed and with the job IDs of any previous successful attempts to compute this job. If a new failure occurs while RCMP is still recovering from a previous one, RCMP’s behavior remains unchanged. RCMP treats this second nested failure like any other failure. It interrupts the currently running job and starts re-computation. This is because RCMP does not need to recover from each failure separately. A re-computation job can service any number of data loss events. RCMP only needs to be careful and tag the submitted re-computation job with the reducer outputs damaged by all failures.

JobInit uses the tagged information to decide which of the persisted data to consider for job re-computation. JobInit checks the metadata on the list of already persisted mapper outputs and readies for execution only the minimum necessary number of mappers. Most persisted mappers are reused. They are treated as if they had already finished. See end of subsection §4.3.3.4 for one subtle exception. Concerning reducers, JobInit readies for execution only the reducers for which the outputs were affected. Note that on re-computation, RCMP departs significantly from current MapReduce systems. Today’s systems do not decide which lost data actually needs to be re-generated because they do not understand the notion of a re-computation job. Today’s systems treat each job submitted to the system as a brand new job and re-execute it entirely.

RCMP performs job re-computations at the granularity of tasks. While it is possible to optimize this further at a per-record granularity we believe that such an approach adds too much complexity to the system. A detailed discussion can be found in subsection §4.4.2.

4.3.2 RCMP During Re-computation

To give a sense of the benefits that RCMP can provide during re-computation by reusing persisted data, consider a system where storage and computation are collocated. After a single node failure, RCMP only needs to compute 1/N of the mappers and 1/N of the reducers. This also translates into a 1/N decrease in the shuffle traffic compared to the initial run of the job. If these 1/N mappers took $W_{M}$ waves in the initial run, they can now
potentially be re-computed in $\text{ceil}((W_M * S)/((N - 1) * S)) = \text{ceil}(W_M/(N - 1))$ waves if they can be distributed over all compute nodes. The same arguments applies for the $W_R$ waves of reducers.

While these are important performance benefits, RCMP’s uniqueness stems from the way in which it efficiently executes re-computation jobs. This efficient re-execution provides additional, significant performance benefits. This subsection details the challenges involved in providing efficient re-computation and the way RCMP tackles them.

### 4.3.2.1 Maximizing Resource Use for Re-computation

Ideally, all available computing resources will be leveraged for re-computations. RCMP’s case is challenging. Because RCMP may end up re-computing a fraction of a job’s tasks there is a real danger that these tasks may be too few to fully utilize all available computing resources.

One may attempt to configure a job so that it is efficient on re-computation. However, this is bound to be inefficient in the failure-free case. For example, to efficiently re-compute after single failures each of the N nodes should run N-1 reducers so that each of the surviving N-1 nodes helps with re-computation. This results in $W_R = (N - 1)/S$ reducer waves in the failure-free case. If $W_R$ is big (e.g. $N = 100, S = 10$) then, in failure-free runs, time is wasted performing a shuffle for each wave of reducers. One can keep $W_R$ small by increasing $S$ (e.g. $N = 10, S = 10$) but then performance is impacted because too many reducers are running concurrently on each node and contend for resources.

The root of the problem lies in the granularity at which tasks are scheduled. A coarser granularity is often desired for the initial run because it simplifies scheduling, offsets costs associated with tasks start-up and shut-down and at the same time uses resources efficiently because many tasks need to be executed. Unfortunately, the same coarse granularity can severely under-utilize computing resources under re-computation when only a few tasks need to be re-computed. To exemplify consider the case when $W_R * S << N$ (i.e the total number of reducers ran by a node for a job is smaller that the total number of nodes used).
In this case, reducer slots will be severely under-utilized during re-computations. After a single failure most nodes (i.e. $N - W_R * S$) will run no reducers. This case is the norm today because it is more efficient to set the number of reducers so that $W_R$ is 1. This allows the shuffle phase to overlap with the map phase [93]. Thus, using the task granularity from the initial run for re-computations has profound negative implications. The job re-computation time, instead of being bounded by the number of available computing resources ends up being bounded by the impact of the failure (i.e. the number of tasks affected by failure) and the job configuration (which dictates the number of tasks per node).

RCMP addresses this resource usage challenge by switching to a finer-grained task scheduling granularity only during re-computations. RCMP effectively balances the benefits of the two types of task granularities using each when it is most efficient. Note that RCMP departs from today’s systems which use a single, static task scheduling granularity defined at job configuration time. Today, since job re-computations are simply entire re-runs of a job, if the initial job was provisioned with the right task scheduling granularity that maximizes resource use, the same will likely hold for its re-computation.

RCMP’s approach is to split tasks that belong to re-computation jobs. We focus on reducer splitting because mappers are less likely to under-utilize resources since they are...
usually more numerous and there is no negative side-effect of having $W_M >> 1$. Nevertheless, mappers can be trivially split since oftentimes each record is processed individually. Reducer splitting works as follows. An initial reducer is responsible for a number of keys. During re-computation the keys are simply divided among the multiple splits of the reducer. Each split still is responsible for all the values belonging to one key and this ensures computation correctness. Users should configure RCMP to split reducers only if the application logic allows it. For example, a reducer performing a top-k computation may not be split. Fortunately, such cases are rare. For simplicity, in our current implementation the split ratio for re-computation is configured to efficiently use the available nodes.

Figure 4.4 illustrates the benefits of task splitting by looking at a re-computation job during which one single reducer (R1) needs to be re-computed. Computation and storage are not collocated in this example. A storage-side failure caused the loss of R1’s output but all mapper outputs (M1, M2, M3) can be reused because they are hosted on compute nodes. In case a) reducer splitting is not used and 2 compute nodes have idle reducer slots. One node has to re-compute R1 entirely. With splitting (case b), the reducer work is divided among all available nodes and each split reducer contributes a portion of R1’s output data. Note that reducer splitting helps not only because it better uses the available compute node parallelism but because it also load balances data transfers across more disks and more network links.

### 4.3.2.2 Avoiding Hot-spots

Under re-computation there is also the danger of encountering hot-spots when many mappers concurrently converge on one storage location to obtain their input data. For illustration, consider the collocated environment in Figure 4.5. Case a) illustrates an initial job run in which one of the nodes, node Y computes reducer R1 and in the subsequent job it compute 3 mappers in 3 different waves, because it has just one mapper slot. These 3 mappers are based on R1’s output. Suddenly, node Y dies and R1 and the 3 mappers (M1, M2, M3) need to be re-computed. During re-computation (case b), node Z re-computes re-
Figure 4.5: Increase contention (hot-spots) on storage during re-computation.

Reducer R1 but the 3 mappers based on R1’s output are re-computed in 1 single wave because they are distributed over 3 surviving nodes. Since they run in one single wave, all mappers will attempt to simultaneously access node Z to get their input, thus severely increasing contention on Z.

To quantify the magnitude of the contention, consider that nodes Y and Z have D local disk drives. During the map phase of the initial run, the average number of mapper accesses on the D drives is on the order of $S$, which is the number of mapper slots on a node. Under re-computation, the contention on node Z can be as high as $S \times N$ which is the number of mapper slots over all available nodes. Furthermore, a network bottleneck may also appear because of the large number of simultaneous transfers.

RCMP can also use reducer splitting to mitigate the hot-spots. This works because reducer splitting distributes the reducer computation over many or all available nodes. Thus, this also implicitly distributes reducer output data over the nodes, mitigating the contention in the subsequent map phase. In effect, reducer splitting helps speed-up both the current re-computation job as well as the subsequent one.

We have implemented and experimented with an alternative solution for mitigating hot-
spots. Specifically, instead of splitting, RCMP can instruct the reducers belonging to re-computed jobs to write each output block to a different storage location in order to distribute the reducer outputs. This solution also balances the mapper accesses in the subsequent job but compared to reducer splitting, it does not have the added benefit of dividing reducer output writing or shuffle work among several nodes. As a result, its capability to lower job running times is reduced, especially when the shuffle phase is significantly more expensive than the map phase. In this case, speeding up just the map phase may not improve overall job running time because the shuffle will still be the bottleneck. This shuffle-bottlenecked case is relevant for RCMP because it may appear when only a small fraction of the mappers need to be re-computed or when the network is slow. In both cases the fundamental cause is that the re-computed reducers need to shuffle data from all mappers, including the persisted ones. The alternative solution does have some benefits. It can be used even when the application logic does not allow reducer splitting and can improve cluster utilization when able to reduce mapper running time.

4.3.3 System Building Challenges

4.3.3.1 Data Access Model

Hadoop and MapReduce are built around a WORM (Write-Once-Read-Many) data access model. To enable efficient failure resilience RCMP allows only portions of a file impacted by failure to be re-computed. This may seem to depart from the WORM model as a file is written again during re-computation. Note that re-computation is transparent to the application so from the point of view of the job running on top of RCMP the data access model is still WORM. Moreover, RCMP does not have to deal with the complexity of an access model that allows multiple writers. In RCMP, the output of the re-computed job is written to a new file. After the re-computed job finishes, RCMP simply manipulates the metadata so that the lost blocks in the affected file are replaced with the blocks from the new file. As there can only be one re-computation in progress for a specific computation, this change of metadata can be safely performed. From the point of view of the job that follows the input
file has not changed at all. It is the same file that existed before the re-computation. The only difference is that the location of some of the blocks may have changed.

4.3.3.2 Choosing the Reducer Splitting Ratio

RCMP can use any reducer splitting ratio but this ratio has to be decided before a job begins executing for the first time. The simplest way to achieve this is to give the splitting ratio as a configuration parameter for the entire computation. This is not a major limitation, as the user can choose the ratio such that under the most probable failure scenario the re-computation uses all the compute nodes that are expected to remain available. Even if the reducer splitting value is not optimally chosen (e.g. more or fewer failures than expected occur) this is not a problem as significant benefits will still be obtained.

We chose to have the reducer splitting ratio set early, as described, in order to limit the changes to the Hadoop code base. Conversely, deciding the splitting ratio at re-computation time requires a non-trivial amount of changes. To explain the difficulty, recall that a completed mapper output is stored in a sorted and partitioned form, with the number of partitions being equal to the number of reducers in that job. This facilitates the data transfers in the shuffle phase as each reducer can directly copy its partition. The challenge that RCMP faces is the following. Assume that during the initial run of a job each mapper output was partitioned into R parts because there were R reducers in that job. Assume that the job is now being re-computed with R1 being the only reducer that is re-computed. Consider also M to be a mapper for which the persisted output is being reused. M’s output was already partitioned during the initial run, and there is one partition ready for R1 to copy. Let that partition be M\_R1. Consider that RCMP decides to split R1 in X splits. If RCMP were to choose a splitting ratio of X just before the re-computation job starts, then each reused partition of a reused mapper output (such as M\_R1) needs to be further re-partitioned into X partitions (one for each split). This is non-trivial to implement and requires additional data accesses and manipulation. The fundamental problem for the system is handling the two different task scheduling granularities: one used for the initial run and one desired for
re-computation. Instead of re-partitioning reused mapper outputs, we implemented RCMP in the following manner. Assume that a job has $R$ reducers and that $X$ is the configured reducer splitting ratio. When a mapper output is materialized to disk it is partitioned in $X \times R$ partitions. This allows easy access to the partitions for both reducer granularities. During the initial run each reducer $i$ asks for the partitions between $X \times i$ and $X \times (i + 1)$. During re-computation each split $k$ of reducer $i$ asks for partition number $X \times i + k$.

4.3.3.3 Re-using Mapper Outputs Correctly

RCMP needs a way to correctly determine which persisted mapper outputs to reuse during re-computations. We have implemented this using mapper IDs. Each mapper has an unique ID that also links it to the job that it is part of. This simple approach works because the mappers in a re-computation job are normally the same as the mappers in the initial job. Unfortunately, challenges can appear when using reducer splitting due to Hadoop’s approach to creating mapper tasks. Basically a re-computation job may create additional mappers compared to the initial job. Key to understanding this behavior are two facts. First, Hadoop creates a different HDFS file for the output of each reducer regardless of whether the reducer is a split or not. Second, Hadoop never creates mappers that read their input from two different HDFS files even if this means that the mapper will end up processing less than the maximum amount of data that it can read (typically the size of an HDFS block).

Figure 4.6 shows an illustrative, simplified example. Reducer R3 and mapper M3 were lost due to failure and are re-computed. During job 1* (re-computation of job 1), R3 is split in 3 (R3.1, R3.2, R3.3). Consider that in job 1 each initial reducer, including R3, outputs one HDFS block. Thus, under re-computation, R3.1, R3.2 and R3.3 will each output roughly 1/3 of an HDFS block. Hadoop will create a mapper (M3, M4 and M5 respectively) for the output of each of the 3 splits.

At the beginning of job 2*, when deciding which mappers to run and which mapper outputs reuse, RCMP sees that M1, M2 and M4 were stored during job 2 and reuses them to
Figure 4.6: The danger of re-using mapper outputs incorrectly when using reducer splitting. The highlighted tasks are re-computed.

Figure 4.7 presents a subtle challenge resulting from the interaction between non-locality, splitting, and data partitioning. In the initial run of job 2, mappers M2.1 and M2.2 each process half the keys that R1 outputed. A failure occurs and the outputs of the grayed tasks (R1 and M2.1) are lost and need to be re-computed. The output of M2.2 survived because M2.2 was a non-local task. During re-computation, R1 is split in two (R1.1* and R1.2*). The keys initially processed

4.3.3.4 When Mapper Outputs Should Not Be Reused

Correctly performing splitting may seem trivial but is not. Figure 4.7 presents a subtle challenge resulting from the interaction between non-locality, splitting, and data partitioning.

In the initial run of job 2, mappers M2.1 and M2.2 each process half the keys that R1 outputed. A failure occurs and the outputs of the grayed tasks (R1 and M2.1) are lost and need to be re-computed. The output of M2.2 survived because M2.2 was a non-local task. During re-computation, R1 is split in two (R1.1* and R1.2*). The keys initially processed...
Figure 4.7: Illustration of a potential problem caused by hash partitioning and reducer splitting. Highlighted tasks are re-computed.

by R1 are now partitioned between its two splits using hash-partitioning. All the keys which modulo 2 yield the same result end up at one of the splits: R1.1* processes the odd keys and R1.2* the even. It may seem that RCMP could reuse the output of M2.2 and not even re-compute R1.2* and M2.2*. This would be incorrect! M2.2 and M2.2* are not the same because of the hash partitioning. Re-using the output of M2.2 would cause keys 11,13,15,17,19 to appear twice in the job output, and keys 2,4,6,8,10 to never appear. RCMP solves this problem by not re-using the mapper outputs (such as M2.2) for which the reducer they depend on has been split during re-computation.

4.4 Discussion

4.4.1 Storage Footprint and Management

To provide efficient failure resilience, RCMP trades off storage space. We consider this to be an acceptable compromise, supported by recent research that shows storage space is plentiful in data centers. For instance, one study finds that on a 240-node cluster, 50% of the files were not accessed in the 250 days prior to the measurement [57]. RCMP can put this wasted storage space to good use to speed-up currently running applications in the face of failures.

A back-of-the-envelope calculation illustrates RCMP’s storage footprint compared to
Hadoop. Assume a multi-job computation which has an input/shuffle/output ratio of 1/1/1 for every job. Assume that the outputs of the intermediate jobs are of no interest to the user so they can be deleted if desired. We do not count the input of the first job since it is the same for both Hadoop and RCMP. Hadoop needs \((2R + 1)x\) storage space where \(x\) is the job input size and \(R\) is the replication factor. \(Rx\) is needed for the input of the currently running job, \(Rx\) for the output of the currently running job and \(x\) for the mapper outputs. For Hadoop, all data from previous jobs can be discarded since Hadoop relies on replication for failure resilience. In contrast, the maximum storage footprint for RCMP is \(2xJ\) where \(J\) is the number of jobs in the multi-job computation. Each of the \(J\) jobs requires \(x\) for the mapper outputs and \(x\) for the job output. If we consider as an example \(R\) to be 3 and \(J\) to be 10, then using the estimations above, the storage footprint for Hadoop is 7\(x\) and for RCMP is 20\(x\). Note that for Hadoop this storage footprint is strictly necessary while for RCMP it is the footprint that provides the most efficient failure resilience. The minimum storage footprint necessary for RCMP to correctly finish the multi-job computation under failures is \(3x\) (input, output and shuffle data for the current job). Unfortunately, this minimum footprint will not provide efficient failure recovery.

Additional storage management capabilities could be added to RCMP to lower its storage footprint. A first approach is to use infrequent replication as a means of bounding RCMP’s storage cost. On re-computation, RCMP would cascade back only until the last replication point and it could discard the data persisted before the replication point. In storage-constrained environments, RCMP would need to more aggressively reclaim the storage space in order to make room for the data required for the job to finish. For this, as future work, we plan to implement an eviction policy that maximizes the speed-up that the remaining persisted outputs bring on re-computation. There are two sides to such a policy. First, it is more efficient to evict persisted data such that under re-computation complete waves of mappers or reducers are executed. Executing an incomplete wave takes the same amount of time as executing a complete wave. Second, not all mapper outputs may need to be persisted. Consider a job for which the shuffle is the bottleneck. Persisting mapper
outputs may not provide benefits because even if the map phase is sped-up, the shuffle will still remain the bottleneck.

4.4.2 RCMP Re-computes Tasks Entirely

When RCMP re-computes a task, it re-computes it entirely. It may split tasks for efficiency but the union of all splits is still the entire initial task. In other words, during the re-computation of a task the same amount of work is performed as in the initial run. With splitting, the difference is that the re-computation work is performed by many machines in parallel instead of using only one machine as in the initial run. However, an optimal implementation may be able to re-compute only parts of the affected tasks. We have experimented with such an approach but saw that the additional complexity brought only limited benefits.

Under a single failure, re-computed mappers would ideally only do a small portion of the initial work (1/N for a balanced computation), strictly the amount necessary for the 1/N re-computed reducers. However, it is difficult to make mappers skip specific input records because the reducer destination of each mapper output record is only decided after the map function is applied to the record. We experimented with doing the next best thing which is disregarding all re-computed mapper output records that are not necessary for re-computed reducers. This reduces the disk I/O necessary to materialize the mapper outputs to 1/N. The benefits that we measured experimentally were limited and did not change either the conclusions of this chapter or the magnitude of RCMP’s benefits.

The same type of optimization could potentially be applied to the reducers. Consider a reducer that writes its output to multiple disks on the local compute node but just one of the disks fails. A good part of the reducer’s output is available on the other disks but RCMP will re-compute the entire reducer task. The reason is that it is difficult for the reducer to know which mapper output records to skip to re-generate only the lost blocks.

A similar situation can also occur under multiple failures when reducer splitting has already been used during the failure recovery for a previous failure. On a first failure, when
using reducer splitting, it is likely that every split will store its data in a different storage location. If a second failure occurs, then it is likely that only the output of one or of a few splits is lost. An optimal implementation of RCMP would re-compute only the affected splits. For simplicity, RCMP currently re-computes all the splits of the initial reducer if any of its splits was affected.

4.4.3 RCMP and the Collocation of Storage and Computation

In this chapter we mentioned two types of cloud environments. The first type is the collocated case where the same set of nodes perform computation and store the data in the distributed file system (e.g. a node is both a storage node and a compute node). The second type of environment is the non-collocated case where storage and computation are separated. RCMP provides benefits in both environments. With respect to RCMP, the fundamental difference between the two environments is the effect of node failures. For the collocated case, node failures cause the loss of both reducer outputs and mapper outputs. For the non-collocated case, a compute node failure causes the loss of persisted mapper outputs and a storage node failure causes the loss of reducer outputs. The problem of inefficient resource usage during re-computation equally applies to the two cases. We discussed hot-spots in a collocated environment in this chapter. Hot-spots during re-computations can equally appear in the non-collocated case. Consider that a compute node fails but no storage node is affected. All mappers on the failed node may need to be re-computed and they can be distributed over the remaining compute nodes. If all mappers obtain their input from the same storage node, then a hot-spot will appear.

4.4.3.1 How Far Back Should Re-computation Cascade On Failure

In the current implementation of RCMP, re-computations cascade back to the first job for which the entire input has survived the failure. Essentially, RCMP re-generates all affected reducer outputs. In most cases this is necessary because each affected job needs the output of the previous job to be complete before it can re-generate its own output.
Figure 4.8: Illustration of how mapper locality can influence how far re-computations cascade back. The large red arrow points to the furthest point to which re-computation will cascade to under a single node failure. Tasks affected by failure are highlighted. Node X runs tasks *.X.

There are however exceptions. A first exception is a non-collocated environment. If a compute node fails, there is no need for cascaded re-computation since the input of the currently running job is unaffected. If a storage node fails, then RCMP may need to cascade back but only until the previous job for which all mapper outputs are persisted.

There is a similar exception for collocated environments. Consider Figure 4.8 and assume that the failure occurs just before job 3 finishes execution. The left side pictures a typical re-computation which cascades back to the beginning of the computation. The right side pictures the case where all mapper tasks are non-local. In this case, RCMP need not cascade at all, unless re-generating the intermediate job outputs is of interest to the user. This is because M3.1 was non-local and therefore its input survived. The input of M3.3 was lost but M3.3 itself was not affected by the failure. As a result job 3 can safely continue simply by re-computing task M3.1. Notice that the likelihood that this scenario occurs unintentionally is small because all mappers ran on the failed compute node have
to be non-local. If even one mapper was local, then that job will not be able to continue until its input is re-generated. As future work we plan to study the benefits of intentionally using mapper non-locality for providing failure resilience and limiting the amount of re-computation necessary on failure.

We have not implemented the afore-mentioned improvements in RCMP because of the separation of duties between the middleware component and the manager. The manager informs the middleware of lost job outputs but offers no information regarding task locality. As a result the middleware has to assume the worst case and cascade back as far as it is necessary in order to re-generate all affected job outputs.

4.4.3.2 Deterministic Computations

As RCMP re-computes parts of previous jobs on failure, a legitimate question is whether the result of the re-computation will be the same as the result obtained during the initial run. Fortunately, this holds for deterministic computations which form the majority of big data jobs. Nevertheless, big data jobs can also involve probabilistic algorithms or statistical computations. For example, one can write an algorithm that produces an estimate of the average employee salary by sampling instead of using all input entries. When tasks that perform sampling are re-run it is conceivable that the result will not be identical with the one from the initial run. This may not always be a problem, as the new execution is just another correct execution of the algorithm. Moreover, the MapReduce model itself relies on task-level re-computation for speculative execution. Therefore, a re-computed MapReduce task is deemed to be identical to the initial task. Nevertheless, complications can appear when moving to job-level re-computation. To illustrate the danger consider a hypothetical algorithm that depending on the output of job X continues with either job Y or job Z. During the initial run, the sequence was job X followed by job Z. Now consider that a failure occurs and job X is re-computed. Because of the re-computation, the result of job X changes and the correct sequence of jobs becomes X, Y. However, RCMP will continue to re-compute following the job sequence in the initial execution and will continue with job Z instead of
job Y. Therefore, RMCP leaves it to the user to decide whether job re-computation will affect the correctness of the computation.

4.5 Evaluation

4.5.1 Methodology and Environment

RCMP is built on top of Hadoop version 1.0.3, the latest release at the time this project was started. HDFS is used as the file system. The implementation of RCMP necessitated roughly 1000 lines of code not counting comments, empty spaces, print and logging statements or Java import statements. In total 37 Hadoop classes needed modifications. We believe that the amount of changes suggest that adding support for job re-computation in Hadoop is a manageable undertaking, worthwhile especially given the significant performance improvements showcased in this evaluation.

What we compare We compare Hadoop 1.0.3 with RCMP. Hadoop uses a replication factor of 2 or 3 (called REPL-2 and REPL-3). With a factor of 1, Hadoop would not survive any failure. RCMP uses a factor of 1 (writes one HDFS replica) since it can recover by re-computing.

The computing environment To evaluate RCMP we used STIC[20], one of Rice University’s compute clusters. STIC nodes have 8-core 2.67Ghz Intel Xeon CPUs and are interconnected by a 10Gbe network. Each compute node has only one local S-ATA drive. We only used this local storage for the experiments, thus our experiments cover the case where computation and storage are collocated. Because the nodes have one local drive and a fast network we expect our experiments to be mostly disk bottlenecked. Nevertheless, we also emulate a network-bottlenecked environment in the experiments in §4.5.6. With the exception of subsection §4.5.5 all other experiments use 10 compute nodes. In all cases, a separate extra node runs the JobTracker and the NameNode. All compute nodes are non-virtualized and we have exclusive access to them. Note that a larger cluster would have been even better for RCMP because in that case the re-computation work can be divided
among more nodes.

**The multi-job computation used** We have built a custom 7-job chain computation. A job starts only after the previous job in the chain successfully finishes. For I/O-intensive jobs which are the use-case targeted by RCMP we believe that the exact work performed by the mappers and reducers is inconsequential. The reason is that these types of jobs are usually bottlenecked by reading/writing data and not by the computation itself.

Therefore, we decided on the following map and reduce UDFs which are lightweight enough to ensure the computation is I/O-intensive but also expressive enough to allow us to verify the correctness of RCMP’s implementation. Each mapper and each reducer implements two checks for correctness. Both checks are based on the value component of each key-value pair record because these values should remain unchanged throughout our computation. The first check simply adds every byte in the record values to one single per-task counter. The second check computes an MD5 hash for every record value and updates a per-task byte array for which a position P contains the sum of all bytes on positions P in all previously computed hashes. By adding these per-task values across all tasks from the same phase of a job one can verify job and computation correctness. Note that these checks were designed to be lightweight and may have false negatives (i.e. they may not catch all problems which can result in incorrect data modification). Therefore, we also implemented and verified RCMP’s correctness with a more heavyweight approach that compares the hashes of every single output record but we did not use this heavyweight approach for this evaluation. Furthermore, each mapper randomizes the key of each record to ensure that for each job the size of the data transfers in the shuffle stage are not skewed by the computation performed in the previous jobs. The randomization is designed such that a re-computation job is identical to the corresponding initial job: each byte in the key is randomized using as seed the sum of the bytes on the previous positions in the non-randomized key. To further characterize the computation performed by the tasks note that mappers can sustain a compute rate of about 6000-7000 records/sec. The experiments in 4.5.6 are the only experiments in which the tasks do not use the correctness checks, but
we believe that the results for those experiments are equally meaningful. The reason is
that for all other experiments we have experimented with mapper and reducer tasks both
with and without the correctness checks and the results showed only differences which are
inconsequential for the insight and the message in this chapter.

Our job has a ratio of input/shuffle/output size of 1/1/1. This ratio is in between the
range of ratios encountered today in practice [28, 36, 37]. Moreover, it is the same ra-
tio used for sorting, a popular barometer of cluster performance. The relative benefits of
RCMP vs Hadoop are expected to increase when the job output is relatively larger com-
pared to the input and shuffle (i.e. ratios of the form $x : y : z$ where $z > y$ and/or $z > x$
encountered in jobs like Pig Cogroup or creating a web index [28]).

The 7-job computation uses 40GB of randomly generated, triple replicated, binary in-
put data obtained by running the Hadoop RandomWriter job. Each record has a key of
random size varying between 10 and 1000 bytes while each record value has a random size
between 0 and 20000 bytes. This input to the first job in the multi-job computation is triple
replicated. Each of the 10 compute nodes processes roughly 16 mappers, each mapper pro-
cessing 256MB (the HDFS blocks size is 256MB). The results that we present are averages
over 5 separate runs of the 7-job computation. The reducer splitting ratio is chosen to use
efficiently the available compute node resources under re-computation.

How the jobs are numbered   Each job that starts running, regardless of whether it is a
re-computation or not, receives as an unique ID the next available integer number starting
with 1. Re-computations increase the total number of jobs ran. For an illustration consider
Figure 4.9 which depicts the different moments at which we injected failures during the
experiments. Consider case c). A failure occurred during the 7th job. As a result, RCMP
re-computes the first 6 jobs and then restart the 7th. In this case RCMP started a total of 14
jobs; each of the different 7 jobs was started twice. On the other hand, since Hadoop uses
replication to tolerate failures it never has to re-start jobs in our experiments and always
starts a total of 7 jobs.

How failures are injected   We inject failures by killing both the Hadoop TaskTracker
and DataNode processes on a randomly chosen compute node. We injected failures 15s after the start of some chosen job. The only exception is when we inject two failures in the same job. Then, the second failure is injected 15s after the first one. Both Hadoop and RCMP have been configured with failure detection timeouts of 30s. This essentially decouples our results from the variations in the efficiency of Hadoop’s failure detection algorithms, variations which we have detailed in chapter §3. Thus, a first failure is detected roughly 45s after the job start.

For RCMP, we chose the moments to inject failures as follows. We do not inject failures during the first job since its input is replicated. Case b) in Figure 4.9 represents a single failure impacting the computation early. RCMP needs to re-compute just 1 job. In case c), the failure impacts the computation when it is close to completion. RCMP has to re-compute 6 jobs. Case d) shows double failures injected early while in case e) the failures are injected when the multi-job approaches completion. Case f) is an example of a nested failure: the second failure occurs while re-computation is still being performed to address the first failure. For Hadoop we inject failures early (job 2) or late in the computation (job
On the efficiency of our implementation  Currently, RCMP has a number of implementation inefficiencies that put it at a slight disadvantage. For instance, for the job during which the failure occurs, RCMP discards the partial results computed before the failure. Thus, currently, the roughly 45s necessary for RCMP to react to one failure are pure overhead. Ideally, RCMP would freeze the affected job, perform re-computation as needed to re-generate the lost data and then reuse the partial results after restarting the frozen job. Hadoop does not suffer from this inefficiency as it uses replication. If we had set the failure detection timeout to more than 30s, or if the failure were injected late in a job (instead of after 15s) then RCMP would be at an even greater disadvantage.

Evaluation road-map  We first present overall system comparisons between Hadoop and RCMP in the absence of failures and under single and double compute node failures. We then break down the results obtained and assess the capability of RCMP to maximize resource use and mitigate hot-spots by using reducer splitting. Finally, we isolate and analyze the contribution of the reduce phase and the map phase in the re-computation speed-up provided by RCMP in both a disk-bottlenecked as well as a network-bottlenecked environment.

Notations and clarifications regarding the figures  The error-bars in the bar-charts show the minimum and the maximum job running time for a set of runs. The bar represents the average job running time. The x-axis notation for the bar charts is as follows. "NO FAIL" means no failure was injected while "FAIL X" means failure injected at job X. "FAIL X,Y" means that a first failure was injected at job X and a second failure was injected a job Y. "NO SPLIT" means RCMP did not split reducers during job re-computations, while "SPLIT 8" means that a reducer split ratio of 8 was used.
Figure 4.10: Hadoop vs RCMP under single and no failures, 1 mapper slot and 1 reducer slot per node. Lower is better.

Figure 4.11: Hadoop vs RCMP under single and no failures, 2 mapper slots and 2 reducer slots per node. Lower is better.

4.5.2 Overall System Comparisons

No failure and single failures  Figures 4.10 and 4.11 present the benefits of RCMP against Hadoop under no failures and single node failures. In both cases the reducers are
ran in a single wave. There is 1 reducer per node in Figure 4.10 and 2 reducers per node in Figure 4.11. Increasing the slot count improves overall performance for both RCMP and Hadoop because resources are better utilized and periods of idleness reduced. Nevertheless, the relative benefits of RCMP persist. Under no failures Hadoop REPL-2 is 30%-40% slower than RCMP while Hadoop REPL-3 is 70%-100% slower. Even when a failure occurs, RCMP remains efficient slightly edging Hadoop REPL-2. Reducer splitting improves performance every time it is used but more so when the failure is injected during the 7th job because more jobs are re-computed and can benefit from splitting.

**Double failures** Figure 4.12 shows the results for double failures. For this experiment we only compare RCMP against Hadoop REPL-3 because REPL-2 cannot protect against all double failure events: some failures may affect both replicas of a block of input data. As it is intuitively expected, Hadoop performs better when the failures are injected late since only a small portion of the computation needs to be executed with fewer nodes (the ones that were not failed). On the other hand, it is challenging to assess beforehand under which sequence of double failures RCMP is most efficient. The trade-off is the following. If the failures occur late (FAIL 7,14), then RCMP needs to re-compute many jobs but after the re-computation is finished few jobs will have to be fully completed with fewer nodes. If the failures occur early (FAIL 2,4), RCMP re-computes few jobs but after that many jobs will have to be completed with the fewer surviving nodes. Therefore, deciding the best and worst cases depends on the speed of re-computation compared to the overhead caused by using fewer compute nodes.

Nevertheless, in all the runs RCMP performs well and consistently beats Hadoop when reducer splitting is used. Splitting benefits case FAIL 7,14 the most (lowers total running time by almost 1000s) because most jobs are re-computed then. Note that RCMP successfully and efficiently handled a nested double failure (FAIL 4,7) where the second failure occurs while RCMP is still performing the recovery for the first failure.

**More failures** To protect against $F$ failures with replication, $F + 1$ replicas are needed. If $F + 1$ replicas exist but fewer than $F$ failures occur then replication was unnecessarily
inefficient. If more than $F$ failures occur, the computation has to be restarted. Thus, setting the right replication factor requires guesswork. In contrast, RCMP can recover from any number of failures while performing the minimum amount of re-computation work necessary.

4.5.3 Mitigating Hot-spots

Figures 4.13 and 4.14 show the negative effects of hot-spots in the experiments when RCMP performs re-computations in Figure 4.10 and Figure 4.11 respectively. We specifically look at the runs in which failures are injected at job 7 for RCMP (last two bars on the right in Figures 4.10 and 4.11) because in these runs RCMP re-computes more jobs and therefore the numbers quantifying the magnitude of the hot-spot problem have more statistical significance. Both figures show CDFs of mapper running time. The negative effect of the hot-spots is visible in both figures but it is more evident in Figure 4.14 where using more mapper slots per node (2 instead of 1) translates into more mappers concurrently attempting to read their input from the same location during re-computation. The
Figure 4.13: Splitting reducers mitigates hot-spots and accelerates mappers from the next job. CDF based on the experiments from Figure 4.10 (1 mapper slot and 1 reducer slot per node) when failures are injected at job 7 for RCMP (last two bars to the right). Lower x-axis values are better.

worse the hot-spots are, the larger the mapper running time becomes. After the failure, 9 compute nodes remain and they all run re-computed mappers. In Figure 4.13 there will be 9 mappers concurrently reading the data from one location since there is 1 mapper slot per node. On the other hand, in Figure 4.14 with 2 mapper slots per node, there will be roughly 16 concurrent mappers (all mappers lost due to failure). For both cases notice that reducer splitting mitigates contention because mapper input accesses end up being balanced well across all the nodes. As a result, in both Figures 4.13 and 4.14 the mapper running time with splitting becomes similarly small with only a few exception in the tail caused by straggler tasks.

4.5.4 Job Re-computation Speed-up

Figure 4.15 and 4.16 show how fast jobs are re-computed in the experiments in Figure 4.10 and Figure 4.11 compared to the initial run of the jobs. Without reducer splitting, simply by re-using persisted outputs, RCMP yields a 2x speed-up at the median in both cases.
Figure 4.14: Splitting reducers mitigates hot-spots and accelerates mappers from the next job. CDF based on the experiments from Figure 4.11 (2 mapper slots and 2 reducer slots per node) when failures are injected at job 7 for RCMP (last two bars to the right). Lower x-axis values are better.

Figure 4.15: CDF of re-computation speed-up for RCMP for the experiment in Figure 4.10. 1 mapper slot and 1 reducer slot per node. Bigger x-axis values are better.
Figure 4.16: CDF of re-computation speed-up for RCMP for the experiment in Figure 4.11. 2 mapper slots and 2 reducer slots per node. Bigger x-axis values are better.

Figure 4.17: CDF of reducer running time for the experiment in Figure 4.10 when the failure is injected at job 7 for RCMP (last two bars to the right). 1 mapper slot and 1 reducer slot per node. Lower x-axis values are better.

However, with splitting the speed-up more than doubles in Figure 4.15 and it increased by
Figure 4.18: CDF of reducer running time for the experiment in Figure 4.11 when the failure is injected at job 7 for RCMP (last two bars to the right). 2 mapper slots and 2 reducer slots per node. Lower x-axis values are better.

roughly 50% in Figure 4.16. To understand this difference in speed-up we also need to look at the running time of the reducers which is depicted in Figures 4.17 and 4.18. After splitting, reducer running times are similar irrespective of the number of slots because the reducer work is balanced across all nodes. However, the initial reducer running time is better for the case with 2 reducer slots since this better utilizes the resources. Thus, the speed-up for Figure 4.16 is lower.

4.5.5 Maximizing Resource Use

In this experiment we vary the number of compute nodes. However, each compute still processes 4GB of data (16 blocks of 256MB each). When a failure occurs, the 4GB on the failed node need to be re-generated. We want to quantify the benefit that RCMP yields by efficiently using the available compute node parallelism for re-computation.

Figure 4.19 presents the results. Without reducer splitting, increasing the number of nodes does not provide great benefit in this experiment. The reason is that one compute
node needs to fully re-compute the reducer that was on the failed node. The rest of the nodes have idle reducer slots. The benefit seen without reducer splitting stems from the fact that the map phase is divided over more nodes and is completed in fewer waves compared to the initial run. For reducer splitting, a split ratio equal to the initial number of nodes is used. This provides significant benefits. With splitting, re-computation is able to benefit more from an increase in the number of nodes, as each node performs a diminishing amount of reducer work.

### 4.5.6 Speed-up from Re-computing with Fewer Waves

Having analyzed the benefits of reducer splitting we now turn to the other important source of speed-up for re-computations under RCMP: the reduction in the number of mapper or reducer waves during re-computation compared to an initial job. We present two cases. One is the disk-bottlenecked environment used so far. In the second case we emulate a bottlenecked network by artificially introducing a 10s delay at the end of each shuffle transfer. This is a realistic emulation because most other transfers performed by the tasks...
Figure 4.20: Speed-up obtained with RCMP from having fewer reducers waves during re-computation compared to the initial run.

Figure 4.21: Speed-up obtained with RCMP from having fewer mapper waves during re-computation compared to the initial run.

are local to the compute node they are running on. Splitting is not used for these next two experiments.
For reducers  First, we seek to understand the contribution of the reduce phase re-computation in the speed-up provided by RCMP for our 7-job computation. To isolate the benefits of the reduce phase re-computation, no mapper outputs are reused. All mappers are re-computed. We vary the number of reducer waves in the initial run (1, 2, 4) by varying the total number of computed reducers (10, 20, 40) and keeping the number of reducer slots to 1. We inject a single failure at job 7. For the re-computed jobs all re-computed reducers (1, 2 or 4) fit in 1 wave.

Figure 4.20 shows the results. Recall that a shuffle phase is performed for every reducer wave but only the first reducer wave overlaps with the map phase. If the network is the bottleneck, the speed-up increases slightly super-linearly with the decrease in the number of reducer waves re-computed. This is because the map phase is insignificant compared to the bottlenecked shuffle phase and thus each reducer wave not performed during re-computation also saves a very expensive shuffle phase. In comparison, on a fast network, the speed-up increases sub-linearly because the map phase is no longer insignificant.

For mappers  In this experiment we seek to understand the contribution of the map phase re-computation in the speed-up provided by RCMP. We isolate the impact of the map phase by having 1 reducer wave both during the initial run and during re-computation. We vary the number of mapper waves during re-computation by varying how many mapper outputs are persisted during the initial jobs. Figure 4.21 shows that if the network is the bottleneck, then no matter how fewer mapper waves execute during re-computation the speed-up does not increase. This is because finishing the map phase faster does not decrease the time necessary to complete the network-bottlenecked shuffle. On the other hand, if the network is very fast, the shuffle finishes shortly after the last mapper output is computed. Thus, in this case, any reduction in the number of re-computed mapper waves also decreases job re-computation time. The result is a slightly super-linear increase in speed-up with a decrease in the number of mapper waves re-computed.
4.6 Related Work

**Failure resilience for big data jobs** Perhaps the closest to RCMP is the work on Resilient Distributed Datasets (RDDs) [96]. RDDs are a general purpose distributed memory abstraction for sharing data in data center applications. RDDs provide fault-tolerance by logging the transformations used to build a dataset and then using this lineage information to guide re-computation on failures. There are several important differences between RDDs and RCMP. First, RDDs are geared towards applications that can fit most of their data in memory while RCMP focuses on the general case where data may not fit in memory and thus needs to be written to stable storage. Second, the lineage information allows RDDs to determine *what* to re-compute on failure. RCMP also needs to determine *what* to re-compute but goes beyond that and also focuses on *how* to re-compute. That is, RCMP is designed to address specific challenges faced when performing re-computations: maximizing resource use and mitigating hot-spots. RDD does not deal with such challenges. Third, we quantitatively analyze the overhead of replication and the benefits of re-computation, while the work on RDDs only briefly mentions that a checkpoint-based fault recovery mechanism like replication would be expensive for their experiments. Note that RDD also mentions the term ”efficient fault tolerance”. However, its meaning is completely different compared to RCMP. RDDs are deemed to provide efficient failure resilience by comparing against solutions that use shared-memory as a distributed memory abstraction which are expensive to provide failure resilience for. In contrast, for RCMP, ”efficient fault tolerance” means recovering as fast as possible from failures.

FTopt [88] is a cost-based fault-tolerance optimizer that automatically selects the best strategy for each operator in a query plan in a manner that minimizes the expected processing time with failures for the entire query. FTopt focuses on three failure resilience strategies. The first is NONE, when no special action is taken. The second is called MATERIALIZE and is similar to replication. The third is called CHKPT. It allows a computation to roll-back to a previous state, but the computation would have to re-compute
everything it has done since check-pointing. FTopt does not provide insights into the benefits and challenges of re-computation.

Parallel recovery on failure can be encountered in a number of storage systems. Both RAMCloud [71] and Flat Datacenter Storage (FDS) [68] stripe data across many storage nodes. When one such storage node fails, the data can be quickly reconstructed by harnessing all remaining storage nodes that had stripes of the data. As many as hundreds of storage nodes having in total thousands of disks can participate in the recovery. In contrast, RCMP uses parallel recovery for computations, using the remaining compute nodes to re-compute part of the previously ran jobs and re-generate the data lost on failure.

**Re-using previously computed outputs** There is also related work on optimizing computations by leveraging outputs previously computed by similar computations. Some big data computations are amenable to such optimizations because they have similarities in computation (shared sub-computations) and similarities in input (same input or a sliding window of the input data). The challenge that this related work tackles is determining and maximizing the opportunities for data reuse. While RCMP also reuses previously computed outputs it does not face the same challenges because under failures it needs to perform the same computation on the same input. However, RCMP goes beyond data reuse and focuses on how to best re-compute data that cannot be reused. At a higher level, RCMP’s focus is also different. RCMP deals with the problem of providing efficient failure resilience for applications while prior work in this area focuses on improvements in performance and storage utilization.

In this space, Nectar [57] is a system that automates and unifies the management of data and computation in data centers. In Nectar, data and computation are treated interchangeably by associating the data with the computation that produced it. Thus, duplicate computations can be avoided by reusing previously cached results. Nectar uses fingerprints of the computation and the input to determine similarity to previous runs. Nectar provides a cache server that allows the lookup of stored entries based on the fingerprints. To provide its functionality Nectar automatically and transparently re-writes programs to cache
intermediate results and to take advantage of the cached results. ReStore [47] offers similar capabilities for workflows of MapReduce jobs.

Incoop [32] is a MapReduce system designed for incremental computations. These computations run repeatedly to handle slight modifications to the input. A common example are search engines which run the same algorithms (e.g. PageRank) on a set of data augmented with recent crawled information [74]. Incoop executes these computations automatically and transparently in an incremental manner making sure it reuses applicable task output data persisted during previous runs. Incoop identifies and tackles a number of challenges that limit the reuse of data at the granularity of task outputs when using the default Hadoop and MapReduce design for incremental computations. To maximize the reuse of mapper task outputs, Incoop proposes a stable partitioning of job input. Without it, a change in the input could shift the input of many mappers making them look dissimilar compared to any tasks computed during a previous run. For the reduce tasks, Incoop leverages a hierarchical aggregation scheme using combiners. This allows data reuse to occur at the level of combiner output which is at a finer granularity compared to the entire reducer output. Otherwise a reducer would have to be fully re-computed just because it is assigned one more key compared to a previous run. Finally Incoop proposes a scheduler that takes into consideration the location of reused outputs when assigning tasks.

Haloop [35] is a MapReduce system that supports efficient processing of iterative jobs. These iterative jobs have two idiosyncrasies: a non-trivial subset of the input data of each iteration is invariant (i.e. remains unchanged across iterations - potential for data reuse) and a termination condition needs to be detected (e.g. the change in the output of two subsequent iterations is below some threshold). Both of these idiosyncrasies cannot be efficiently supported under the default MapReduce programming model as MapReduce treats each job independently. Haloop proposes that mappers based on the invariant input be processed only once, in the initial iteration, and their output be cached on nodes that run reducers. Haloop also proposes a new scheduling mechanism that tries to run reducers with the same ID from different iterations on the same node. This ensures the effectiveness of
caching mapper outputs at nodes running reducers and also ensures that reducer outputs can be compared locally and distributedly to search for the termination condition. This is also a form of task output reuse as Haloop obviates the need for an additional MapReduce job to compare the outputs of the last two iterations in order to detect the termination condition.

**Improving the I/O efficiency of big data jobs** By providing ways to efficiently deal with failures when they occur, RCMP obviates the need for using expensive I/O to provide proactive failure resilience. Thus, RCMP improves the overall I/O efficiency of big data jobs. The related work in this area is complementary to RCMP. It improves the I/O efficiency for different phases of a MapReduce job. The existence of these related ideas further amplifies the need for the benefits provided by RCMP. If all the other phases of a MapReduce job are highly optimized, then the relative overhead of using replication as a failure resilience strategy becomes even larger.

Rhea [55] optimizes the map input phase for jobs that selectively use input. Rhea uses static program analysis techniques to detect which data will be accessed by a particular job. Based on this analysis Rhea creates storage-side filters that attempt to discard as early as possible the input records that the computation will not use. ThemisMR [77] optimizes the internal execution of mapper and reducer tasks. ThemisMR ensures that only the minimum number of I/O operations (one read and one write) are performed for each record. It does this with the help of a custom memory manager that allows it to process records completely once they are read without resorting to swapping or writing spill files. ThemisMR also incorporates I/O optimizations designed to minimize seeks and deliver near-sequential disk I/O to applications. Camdoop [72] optimizes the shuffle phase by performing in-network aggregation.

### 4.7 Chapter Summary

RCMP is a system that uses efficient re-computation as a first-order failure resilience strategy for big data analytics. RCMP is geared at multi-job I/O-intensive computations. It
leverages previously persisted outputs to speed-up re-computed jobs but more importantly, during re-computations, it ensures that compute node parallelism is maximized and hot-spots are mitigated. Not using data replication makes RCMP significantly faster during failure-free periods but more importantly, by efficiently performing re-computations, RCMP is competitive even when impacted by single and double data loss events.
Chapter 5

Future Work

5.1 Combining the Benefits of Re-computation and Replication

In chapter §4 we showed that RCMP can provide significant benefits for data-intensive jobs, due to the use of job re-computation instead of data replication as the primary failure resilience strategy. Nevertheless, the use of job re-computation is not without limitations. A first limitation is that for long chains of data-intensive jobs, the penalty paid under failure can be significant because RCMP may have to cascade back to the start of the computation and re-compute parts of all jobs started before the failure. A second limitation concerns the diversity of the computations that re-computation can provide significant benefits for. In general, not all big data computations are purely data-intensive but can actually include a combination of compute-intensive and data-intensive jobs. Since a compute-intensive job spends most of its time processing data, the relative overhead of using data replication for compute-intensive computations may not be necessarily high.

Therefore, in general, depending on the characteristics of the computation it may be less clear whether or not re-computation is a substantially better alternative compared to replication. Other job characteristics can tip the balance between replication and re-computation as well. One example is the size of the input and output datasets. For example, a job that reads far more data than it writes is a good candidate for replication, as replication will be relatively cheap. Lastly, the characteristics of the environment can make either replication or re-computation more or less appealing. For example, an oversubscribed network makes data replication expensive and thus undesirable.

For a general-purpose system it therefore makes sense to combine the benefits of replication and re-computation. In such a system, replication can serve as a way to bound
re-computation overhead both in terms of running time as well as storage space used. For example, under common failure scenarios, cascading re-computations may revert only until the last replication point and not all the way to the beginning of the computation. The interesting research question is how to build a system that automatically chooses when to replicate job outputs. In other words, to automatically decide when to use replication and when to use re-computation as a failure resilience strategy. The challenge is that the system would have to make a cost-based decision while using incomplete or limited information about the characteristics of the subsequent jobs in a multi-job computation. The danger is that a decision that seems beneficial at some moment in time may turn out to be clearly suboptimal after a few more jobs are executed.

In its current implementation, RCMP does not decide the outputs of which jobs should be replicated. RCMP simply follows the job configuration provided by the user. It assumes that the user has the corresponding knowledge and configures jobs accordingly. Alternatively, a simple static approach that replicates job outputs once every constant number of jobs can easily added on top of RCMP. To allow RCMP to dynamically decide when to replicate, additional functionality is needed.

First, RCMP needs to estimate the time required to replicate the output of the current job. For this, it is important to have a good estimate of the job output size. If imbalances in the task-level output size are possible, then RCMP would have to wait until all tasks of a job finish before deciding if to replicate the job output or not. If imbalances are not a concern, then RCMP can estimate job output size based on the task output size and decide whether to replicate or not as soon as the first task finishes. Once the job output size is estimated, RCMP would have to compute an estimate of the time necessary to replicate the data. A first potential approach is to use historical information about previously performed replications in the context of the same job or other jobs.

Second, RCMP needs to estimate the cost of potential re-computations. To do this, RCMP needs to estimate the probability of failures as well as their impact. Historical, cluster-wide failure statistical information can be useful as well as more precise informa-
tion about the specific compute nodes involved in the computation as studies show that when a machine loses the output of a task, it has a higher chance of losing the output of other tasks [27] as well. Using the failure information, the number of available compute nodes and the number and characteristics of the tasks that should be re-computed under the expected failure cases, RCMP can estimate the cost of re-computation for each completed job.

Third, to further optimize replication decisions, RCMP can benefit from knowledge about future jobs in the multi-job computation. One way to estimate such information is via static job analysis [55, 60]. Another approach is to first focus on recurring computations since for these cases information is readily available from previous runs of the computation. Prior work suggests most business-critical computations are indeed recurring [50] jobs. Of great benefit would be information regarding the expected input to shuffle to output ratio of future jobs. This information may significantly increase the efficiency of replication decisions. For example, based solely on past information the system may decide to replicate the output of some job at the current point. However, it is entirely possible that this decision is deemed suboptimal when information about future jobs is incorporated in the decision process as well. To exemplify, consider the case when there is information that the next job to be computed is expected to output significantly less data than the current job. It makes sense to delay replication until that point since the output of the next job can be replicated at a much lower cost.

5.2 Decoupling Failure Resilience from Job Execution

We have argued that data replication is an expensive operation in the context of data-intensive jobs. What makes replication expensive are not only the large datasets involved and the high resource utilization necessary to replicate the datasets. An important fact is that replication is performed as part of the running time of the job. Essentially, current MapReduce implementations couple proactive failure resilience with job execution. Decoupling the two by performing replication in the background can reduce the impact on the
job running time while still providing failure resilience. Moreover, this can allow dynamic replication decisions based on the results of job execution. One example where this can be useful is a computation for which the output needs to converge to some value before completion is declared. Assume that the efficient approach is to replicate only the output of the final job. Unfortunately, for this computation the fact that a job is the last one in the computation is not known a priori but rather only after the job is executed and its output evaluated. Current MapReduce implementations like Hadoop cannot easily handle this situation since they already set the replication factor before the start of the job.

Performing replication in the background is particularly useful when there is idleness in the system. This is not uncommon in current data centers. Several studies suggest disk and network bandwidth are either underutilized or present idleness because of a bursty pattern of usage [26, 36]. The interesting question in this type of environment is whether current idleness patterns allow meaningful failure resilience guarantees to be obtained solely by using background replication. For example, can background replication ensure that the output of any job except the last two is always replicated?

5.3 Improving Speculative Execution Decisions

5.3.1 Better Straggler Detection

Recent work on speculative execution has tried to detect stragglers early and reliably [27, 97]. These approaches essentially detect stragglers by comparing a task’s behavior (usually progress rate) with the behavior of similar tasks belonging to the same job. However, this comparison-based approach can backfire when tasks or environments are not homogeneous. In this case, it is natural for tasks to progress at different rates. Also, this comparison based approach cannot handle situations where a task has no peers to compare against (e.g. a Hadoop job that has a single reducer). Moreover, the approach may not detect stragglers fast since some time needs to pass before a task becomes evidently slower than its peers. The reason is that there always exists some natural variation in execution
speed between tasks and the slow task needs to stand out compared to this variation. In addition, in chapter §3 we showed that a statistical approach can be surprisingly inefficient even in homogeneous environments because task progress rates can vary due to different data availability. An initial reducer may have to wait for mapper outputs while a duplicate scheduled after the completion of the map phase need not wait at all.

An ideal solution to the straggler detection problem would forgo comparisons altogether and would know for each task how fast it should progress before it is even started. Such a solution would be more robust, more general and would react faster to incipient signs of straggling. One starting point is to collect statistics about the task input and test all code paths in the task using representative input before the task is even started. Care needs to be taken when the computation uses state that is shared between the processing of different records as this state can influence control flow decisions in the task code. Unfortunately, this type of testing and the program analysis necessary can be time consuming which is especially detrimental for interactive jobs. The challenge becomes how much benefit can be obtained using a lightweight approach? To enable such a lightweight approach, first, statistics need to be gathered about the output data as part of a job’s regular run. Then these statistics should be plugged into a model of a task’s complexity. How to quickly obtain such a model is an open problem.

5.3.2 Better Straggler Re-execution

Once a straggler is detected, the system should ideally re-execute (restart or duplicate) it as fast as possible. Current speculative execution algorithms [27, 97] only try to ensure that the re-execution of a straggler avoids well known performance problems like slow nodes or congested network locations. This however, is insufficient. The execution time of the duplicate is still bounded by the granularity of the task and the resources present on the compute node it executes on (CPU, network). In other words, duplicates receive no special treatment, a decision in stark contrast with the urgency required in dealing with such tasks that are behind schedule and slow down the entire job.
We propose to leverage the task splitting functionality in RCMP to help execute duplicates significantly faster than regular tasks. In effect, duplicates will be using a more fine-grained task scheduling granularity compared to the initial tasks. This will allow them to divide computation and I/O across multiple nodes. A first challenge for this approach is deciding the splitting factor. On the one hand a low factor may limit speed-up. On the other hand a large factor may increase contention (often leading to performance degradation) and the relative overhead of task start-up and shut-down costs.

5.4 Towards a Cloud Knowledge Plane

In chapter §3 we showed that significant performance degradation can appear due to the lack of sharing potential failure information among Hadoop tasks. We believe that sharing failure information is just a first step. We envision sharing broader types of information and doing this beyond the confines of a single job. In essence, we envision a cloud knowledge plane which can become a fundamental building block for entire cloud applications running in a multiplexed cloud environment.

Multiplexed, public cloud environments are popular today for economical reasons. However, managing the multiplexed cloud environment and the applications in order to achieve the best application performance is challenging. The task is already made difficult by the highly dynamic cloud environment, its unprecedented scale and the large number of hosted applications. Unfortunately, today’s approaches to cloud application management are also limited in the way they acquire information. The operator either performs monitoring in isolation [59] or shares only some basic infrastructure performance data with the applications [1]. The expressiveness of the operator’s data is limited by the overhead of monitoring and it is typically low-level, application agnostic data. Applications are also limited in their self-management capabilities by the operator’s tenant isolation mechanisms which only allow a highly abstracted view of the environment [40].

We believe that multiplexed cloud environments give applications a unique opportunity to cross-share their experiences for potential improvements in performance, security and
scalability. Instead of preventing sharing as is the case today, the cloud operator should arbitrate, facilitate and even participate in the exchanges. We argue that this is a win-win situation for both parties. For the operator, the advantages are improved overall cluster utilization and the access to the shared information which is expected to be fresh and have strong statistical significance. For the applications, the advantages are that performance will likely improve, and that will be able to make decisions regarding resources that have not yet been in their possession but have been used by other applications.

A primary challenge when sharing application experiences is determining whether information shared by one application is applicable to another. The danger is that even small differences among applications (e.g. use of delayed ACK) can significantly influence perceived performance [94]. Fortunately, the cloud presents us with a unique opportunity. Many cloud applications are built on top of a few general-purpose large scale computing frameworks (e.g. MapReduce) which provide the basic functionality of interacting with the environment and managing running jobs. Additionally, the cloud computing environment (i.e. OS, hardware) is completely known to the operator and is usually limited in diversity (e.g. small number of OS instances). Consequently, our position is that many cloud applications have substantial functionality in common thus making sharing experiences in the cloud a promising direction.

Importantly, operator monitoring alone is not as powerful as sharing application experiences. First, for the shared information to be meaningful, the design, configuration and implementation of the measuring component needs to be similar, if not identical. While this may be the case for two MapReduce applications the operator cannot provide different implementations for its measuring infrastructure to suit every application. Second, it is hard for an operator to provide framework specific information (e.g. throughput from dedicated storage to multiple concurrent readers). Third, any large operator monitoring effort uses valuable resources and potentially degrades user perceived performance.
Chapter 6

Conclusion

The goal of this thesis is to enable efficient failure resilience for big data frameworks. By efficient, we mean minimizing the impact of proactive failure resilience algorithms during failure-free periods as well as minimizing the impact of failure detection and recovery during failures. Efficient failure resilience is an important goal because it allows for a more predictable user experience (more predictable waiting times for job completion as well as more predictable costs). Towards our goal we have made two contributions.

We have shown that failure detection and recovery in the popular, state-of-the-art Hadoop big data processing framework is far from efficient. Even a single node failure can result in inflated, variable and unpredictable job running times. We have discovered and analyzed the causes behind the inefficiencies. The causes relate to different design decisions in Hadoop in which efficient failure detection and recovery is traded off for simplicity or scalability.

A first cause is that, for simplicity, Hadoop was built to have little visibility into the infrastructure. As a result it uses connection failures as a proxy for task failures but without a way to pinpoint the cause of the failure this strategy introduces false positives into the system. The second cause is that the speculative execution algorithm in Hadoop ceases to detect slow tasks reliably under failures. The reason is that the statistics it is based on get skewed by fast advancing tasks that appear due to failures. Third, failure detection in Hadoop is a function of the current computation state since currently running tasks help detect failures. This causes variable detection time as the current computation state naturally varies during the running time of a job. Fourth, for scalability reasons, Hadoop tasks do not exchange potential failure information among each other. As a result failures are
individually discovered by each task at a great cost for overall job running time.

As a second contribution we have presented RCMP, a MapReduce system that uses job re-computation instead of data replication as the main failure resilience strategy. RCMP is based on the insight that data replication can have a significant negative impact on the running time of the popular category of data-intensive multi-job computations.

We have explored the challenges inherent in providing failure resilience using job re-computation. A first challenge is re-computing only the minimum amount of work. RCMP does just this by re-computing only the task for which the output was lost due to failure. Interestingly, we have found that simply re-computing the minimum number of tasks yields only moderate benefits.

We have discovered that to obtain the full benefits of job re-computation the system needs to specifically address the manner in which this minimum number of tasks is re-computed. We identified two challenges in this respect. First, the available compute node parallelism may not be well utilized under job re-computation because the number of re-computed tasks may be too small. Second, hot-spots may appear when many re-computed tasks concurrently get their input from the same storage location. We have addressed both challenges by proposing that the system adapt the task scheduling granularity and switch to a finer granularity under job re-computation.

We believe that this thesis paves the way for several innovations in the design of core components on today’s big data frameworks. We have established the need for more robust speculative execution algorithms. We have also shown and quantified the negative effects of making decisions based on scarce environmental information. While good for maintaining overall system simplicity these decisions lead to significant performance degradation. Therefore, there is a need to consider more elaborate methods of collecting environmental information which can lead to quicker and more accurate decision making. Finally, this thesis analyzed the challenges and the benefits that can be obtained from using job re-computation as a failure resilience strategy for big data computations. We have shown that even in isolation this strategy can be very efficient. This paves the way for the design of
hybrid failure resilience strategies that, depending on the workload and environment, can leverage the advantages of job re-computation alongside other failure resilience strategies.
Bibliography


