An Experience Report on Scaling Tools for Mining Software Repositories Using MapReduce

Weiyi Shang, Bram Adams, Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
School of Computing, Queen's University
Kingston, Ontario, Canada
{swy, bram, ahmed}@cs.queensu.ca

ABSTRACT

The need for automated software engineering tools and techniques continues to grow as the size and complexity of studied systems and analysis techniques increase. Software engineering researchers often scale their analysis techniques using specialized one-off solutions, expensive infrastructures, or heuristic techniques (e.g., search-based approaches). However, such efforts are not reusable and are often costly to maintain. The need for scalable analysis is very prominent in the Mining Software Repositories (MSR) field, which specializes in the automated recovery and analysis of large data stored in software repositories. In this paper, we explore the scaling of automated software engineering analysis techniques by reusing scalable analysis platforms from the web field. We use three representative case studies from the MSR field to analyze the potential of the MapReduce platform to scale MSR tools with minimal effort. We document our experience such that other researchers could benefit from them. We find that many of the web field's guidelines for using the MapReduce platform need to be modified to better fit the characteristics of software engineering problems.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Performance

Keywords

Mining Software Repositories; Cloud computing; MapReduce

1. INTRODUCTION

The Mining Software Repositories (MSR) field recovers and studies data stored in large software repositories, including source control repositories, bug repositories, archived communications, deployment logs, and code repositories [23]. The MSR field is one of the many fields within software engineering that continue to benefit from the development of automated software engineering tools and techniques.

These automated techniques continuously need to scale as larger systems are being analyzed and more complex techniques are being used to analyze these systems. For instance, recent studies show that the Debian Linux distribution doubles in size every two years (currently at 323M SLOC [4,17]), while recent efforts [2, 11, 30] continue to archive very large repositories of source code based on the strong belief that the wisdom of thousands of coders can help improve the quality of any one project. Moreover, complex model checking and analysis techniques continue to be developed to help locate bugs in large software (e.g., [38]).

To cope with the scale of the analyzed data and the complexity of the used algorithms, researchers often make use of one-off solutions, heuristic-based optimizations (e.g., search based software engineering [20]), or specialized commercial systems (e.g., [8]). However, these solutions are too expensive to acquire or maintain, and they often require lengthy development time. The lack of off-the-shelf ways to scale analysis techniques hinders research progress, as researchers spend considerable time tackling side-problems that are of limited interest to them, but which they must solve to ensure the adoption of their research in practice. For example, D-CCFinder [28], a distributed version of the CC-Finder [26] clone detection tool, achieves a speed-up factor of 20 using a custom client/server architecture consisting of 80 PCs. This specialized architecture requires substantial development and maintenance effort to keep it running correctly.

Standard platforms are needed that would enable large-scale studies with minimal effort and without the need for continuous maintenance. Over the past decade, the web field has developed a significant expertise in dealing with large-scale problems. That community has developed several standard platforms that have been extensively reused by its members. Hadoop [39] and Pig [33] are examples of such platforms. We firmly believe that our community can benefit from these platforms to scale software engineering studies. In prior work [35], we explored the use of Hadoop, a MapReduce [15] implementation, to scale and speed-up one particular software evolution study. We could reduce the running time of the study tool by 60 to 70%. It is not clear, though whether findings generalize to other kinds of software engineering analyses, and how to address the challenges we encountered with configuration and designing MapReduce strategies.

In this paper, we use the MSR field, as a sub-field within

ASE'10, September 20–24, 2010, Antwerp, Belgium.

software engineering, to study the benefits and challenges of scaling several software engineering analyses using the large-scale data processing platforms. In particular, we use three representative case studies from the MSR field to demonstrate that the MapReduce web analysis platform could be used to successfully scale MSR tools with minimal effort. The main contributions of our paper are as follows:

1. We document our experience in scaling several MSR problems, such that other researchers could benefit from our experience.
2. We also report the changes needed to the web field's guidelines for the MapReduce platform when applying MapReduce to MSR analyses. These changes highlight the different characteristics of software engineering analyses compared to web analyses and must be addressed to ensure that software engineering researchers get the most benefit out of the MapReduce platform.

While we apply scalable web analysis platforms in the context of MSR analyses, we believe that many software engineering research problems that require automated analysis would benefit from these platforms. We hope that our work will encourage other researchers to explore the scaling of their automated techniques using such platforms.

The rest of the paper is organized as follows. Section 2 provides the background and related work of our research. MapReduce and the expected challenges of migrating MSR tools to MapReduce are introduced in Section 3. We present our case study in Section 4, followed by a report about our experiences in addressing the challenges in Section 5. Section 6 evaluates the ability of MapReduce to scale different types of MSR analyses. Section 7 discusses threats to validity. Finally, Section 8 presents the conclusions of this paper.

2. BACKGROUND

Trends in MSR. In recent years, two major trends can be observed in the MSR field that are also representative for many other fields of ASE. The first trend is that the data analyzed by software engineering researchers is exploding in size. Recent empirical studies exhibit such a trend, with many researchers exploring large numbers of independent software products instead of a single software product. Empirical studies on Debian GNU/Linux by Gonzalez-Barahona *et al.* [17] analyze up to 730 million lines of source code from 6 releases of the Debian distribution, which contains over 28,000 software packages. Similarly, Mockus and Bajracharya *et al.* have been developing methods to amass and index TBs of source code history data [11, 30]. Estimation indicates that an entire year of processing is needed to amass such large source code [30]. This growth of data is not exceptional. Studies show that the Debian distribution is doubling in size approximately every two years [17].

A second trend in software engineering is the use of ever more sophisticated automated techniques. Clone detection techniques are examples of this trend. Text-based and token-based techniques, such as CC-Finder [26], use raw source code or lexical "tokens" to detect code clones in a software project. However, as these clone detection techniques are only able to detect a limited number of clone types [34], more complex techniques that require much more computing power and running time are needed to detect more types of code clones with higher precision.

Approaches to scale MSR. The growth of data and the increase in the complexity of MSR studies bring many challenges that hinder the progress of the MSR field. Yet, there is little work that aims to address these challenges.

To enable large-scale MSR studies, researchers continue to develop ad hoc solutions that migrate MSR studies to distributed computing environments. The simplest and most naive way is using batch scripts to split input data across a cluster of machines, deploy the tools (unchanged) to the machines, run the tools in parallel, and finally merge the output of every machine. However, naive approaches mostly do not support load balancing, error recovery and require additional programming effort. Other approaches, such as D-CCFinder [28], Kenyon [14] and SAGE [16], re-engineer the original, non-distributed MSR study tools to enable them to run on a distributed environment. Distributed computing libraries, such as MPI [19], can assist in developing distributed MSR study tools. However, the re-engineering of existing tools requires additional programming effort and software engineering researchers are neither experts in distributed system programming nor willing to spend effort on the programming.

Over the past 20 years, parallel database systems, such as Vertica [8], have been used to perform large-scale data analyses. Recently, work by Stonebraker *et al.* [37] shows that parallel database systems are challenging to install and configure properly and they typically do not provide efficient fault tolerance. MSR researchers are neither experts in installing parallel databases nor can they afford the time to learn the intricacies of such systems. Moreover, MSR experiments typically only extract and read large amounts of data from software repositories, without ever updating this data. Using parallel database system is not an optimal solution for scaling MSR experiments.

Search-based software engineering (SBSE) [21] holds great promise for scaling software engineering techniques by transforming complex algorithms into search algorithms, which yield approximate solutions in a shorter time span. For example, Kirsopp *et al.* [27] use various search algorithms to find an accurate cost estimate for software projects. In addition to optimized performance, most search algorithms are naturally parallelizable to support even larger scale experiments [20]. However, SBSE only offers a set of general techniques to solve problems, and still require considerable application-specific customization to achieve significant speed-ups. Not all MSR analyses benefit from approximate solutions either.

The web field has developed large-scale data analysis platforms over the years. These platforms, such as MapReduce [15], are designed to run in distributed environments, and typically leverage a distributed data storage technique. Widely and successfully used in the web field, these platforms are able to analyze massive amounts of web data. Because the intensive analyses in the MSR field and the web field are both scan-centric (no random access) and read-only, these platforms are very promising for scaling MSR analyses. Shang *et al.* [35] have presented preliminary results that demonstrate that the analysis of large-scale software engineering data could benefit from such large-scale data analysis platforms, despite a number of challenges. This paper explores whether MapReduce can successfully scale a range of typical MSR analyses, discusses how to address the various challenges of migrating MSR analyses to MapReduce

and analyzes the differences between guidelines from the web field and our experience. We then show the applicability of MapReduce in scaling other types of MSR analyses.

3. MAPREDUCE

MapReduce is a distributed platform for processing very large data sets [15]. The platform, originally proposed by Google, is used by Google on a daily basis to process large amounts of web data.

MapReduce enables a distributed divide-and-conquer programming model. The model consists of two phases: a massively parallel "Map" phase, followed by an aggregating "Reduce" phase. The input data for MapReduce is broken down into a list of key/value pairs. Mappers (processes assigned to the "Map" phase) accept the incoming pairs, process them in parallel and generate intermediate key/value pairs. All intermediate pairs having the same key are then passed to a specific Reducer (process assigned to the "Reduce phase"). Each Reducer performs computations to reduce the data to one single key/value pair. The output of all Reducers is the final result of a MapReduce run.

An Example of MapReducing an MSR analysis.

To illustrate how MapReduce can be used to support MSR, we consider performing a classical MSR analysis of the evolution of the total number of lines of code (#LOC) of a software project. The input data of this MSR analysis is a source code repository. The repository is broken down into a list of key/value pairs as *"version number/source code file name"*. Mappers accept every such pair, count the #LOC of the corresponding source file and generate as intermediate key/value pair *"version number/#LOC"*. For example, for a file with 100 LOC in version *1.0*, a Mapper will generate a key/value pair of *"1.0/100"*. Afterwards, each list of key/-value pairs with the same key, i.e., version number, is sent to the same Reducer, which sums #LOCs in the list, and generates as output the key/value pair *"version number/SUM #LOC"*. If a Reducer receives a list with key *"1.0"*, and the list consists of two values *"100"* and *"200"*, the Reducer will sum the values *"100"* and *"200"* and output *"1.0/300"*.

The MapReduce platform holds great promise for scaling MSR experiments, because it is

1. **a mature and proven platform.** MapReduce is widely used with great success by the web field and other communities. For example, the New York Times has recently used MapReduce to transform all its old articles into PDF format in a cost-effective manner [6].
2. **a simple and affordable solution.** MapReduce uses a simple, distributed divide-and-conquer programming model. MapReduce can be deployed on commodity hardware, which makes scaling MSR experiments more affordable.
3. **a read-optimized platform.** MapReduce is designed to perform large-scale read-only data analyses, such as the scan-centric MSR analyses.

Challenges of MapReducing MSR analyses

Although MapReduce holds great promise for MSR, we envision a number of important challenges based on our previous experience of using MapReduce [35]. We use the MapReduce example above to motivate and explain these challenges. The goal of this paper is to document our experiences addressing these challenges across various types of MSR analyses and to carefully examine the guidelines proposed by the web field regarding these challenges. By documenting the differences in analyses and data processed by both communities, we hope that the software engineering field will be able to exploit the full power of MapReduce to scale software engineering analyses.

Challenge 1: Migrating MSR analyses to a divide-and-conquer programming model.

The first challenge is to find out how to migrate an existing MSR analysis to a divide-and-conquer programming model. This migration has two important aspects.

1. **Locality of analysis.** A Divide-and-conquer programming model works best when the processing of each broken data part is independent of the processing of the other parts (i.e., a local algorithm). Counting the number of lines of code (#LOC) for every source file is an example of a local algorithm as this can be done for each file in isolation and the results of each data part can just be added up. Global algorithms (e.g., clone detection [34]) would require each data part (e.g., set of files) to have access to the whole data set. Semi-local algorithms (e.g., source code differencing) require more data than just local data, but not the whole data set (e.g., only two files). It is interesting to note that an analysis might be global due to the implementation of an analysis, not due to the analysis itself. For example, several analyses require access to the full code base, when robust techniques such as island parsing [31] could be used to overcome this implementation requirement and would ensure local analysis.
2. **Availability of source code.** Having access to the source code of an MSR study tool provides more flexible ways to map an MSR algorithm to a divide-and-conquer programming model. However, re-engineering a tool internally increases the risk of introducing bugs.

Challenge 2: Locating a suitable cluster.

Distributed platforms typically run on a cluster of machines. We list below a few aspects for locating clusters:

1. **Private cluster versus Public cluster.** A public cluster is available and accessible to everyone, whereas a private cluster is not.
2. **Dedicated cluster versus Shared cluster.** Dedicated clusters ensure that only one user uses the machines at the same time, while machines in the shared cluster may be used by many users at the same time.
3. **Specialized cluster versus General-purpose cluster.** Specialized clusters are designed and optimized for MapReduce (e.g., [1]), while general-purpose clusters might result in sub-optimal performance.

On the one hand, private, dedicated, specialized clusters provide the most optimal performance. On the other hand, public, shared, general-purpose clusters require the lowest financial cost. There are eight possible combinations of the three aforementioned aspects. To illustrate the possible types of clusters, we show four types as examples.

- **Machines in a research lab (Private, Dedicated and Specialized).** Research shows that computers are idle half of the time [10]. By bundling these computers together, a small cluster can be created.
- **Machines in a student lab (Private, Dedicated and General).** Computers in student labs of universities can be used as medium-sized MapReduce clusters.

- **Scientific clusters (Public, Shared, and General).** Some scientific clusters, e.g., SHARCNET [7], have hundreds or thousands of machines and are specifically designed for scientific computing. The large scale of these clusters enables running experiments on massive amounts of data.
- **Optimized clusters (Public, Dedicated and Specialized).** Some clusters are optimized for MapReduce, e.g., the EC2 MapReduce instances offered by Amazon [1]. Optimized clusters are often too costly.

Challenge 3: Optimizing MapReduce strategy design and cluster configuration.

The different implementations and configurations of the MapReduce platform influence the performance of MapReduce experiments, yet finding the optimal implementation and configuration is challenging.

1. **Static breakdown of analysis.** The optimal granularity for breaking down the analysis should be carefully examined. For example, counting the #LOC of a software project can be decomposed into different data parts that are executed in parallel to count the #LOC of: 1) every source code file (fine-grained) or 2) every subsystem (coarse-grained). The finer the granularity, the more parallelism that can be achieved. However, finer granularity leads to more overhead since additional "Map" and "Reduce" procedures must be scheduled and executed. Although this granularity principle is well known in distributed computing, choosing the best granularity in the context of the MapReduce platform is still challenging.
2. **Dynamic breakdown of processing.** Once the static breakdown is determined, the granularity of processing the input data can still be altered dynamically. MapReduce implementations typically allow sending a number of "Map" and "Reduce" procedures to a machine at the same time as a "Hadoop task". In our #LOC example, one single source code file could be sent to a machine for analysis, or an ad hoc group of files could be sent together in a batch. The composition of Hadoop tasks can be completely arbitrary by the MapReduce platform.
3. **Determining the optimal number of machines.** A third way to optimize the performance of a MapReduce cluster is by changing the number of machines. Adding more machines might not always lead to better performance or effective use of resources, due to platform overhead. For example, adding more machines requires more data transfer over the network, extra computing power, and possibly additional usage fees.

Challenge 4: Managing data during analysis.

MapReduce needs a data management strategy to store and propagate large data fast enough to avoid being a bottleneck. Two data storage choices are typically available:

1. **Distributed file system.** Input data and intermediate data are stored in one distributed file system that spreads its data to every machine of the cluster to increase I/O bandwidth and the total amount of storage, and to achieve fault tolerance.
2. **Local file system.** Saving data in the local file system does not require data replication and transfer on the network.

Table 1: Eight types of MSR Analyses.

Name	Description	Locality
Metadata analysis	Direct analysis on the extracted metadata from software repositories, e.g., [13].	local
Static source code analysis	Static program analysis on source code, e.g., [18] .	local/global/ semi-local
Source code differencing and analysis	Analysis of changes between versions of source code, e.g., [22].	semi-local
Software metrics	Measuring and analyzing metrics of software repositories, e.g., [36].	local/global/ semi-local
Visualization	Visualizing information mined from software repositories, e.g., [32].	global
Clone detection methods	Detecting and analyzing similar source code fragments, e.g. [26].	global
Data Mining	Applying Data mining techniques on software repositories, e.g., [29].	global
Social network analysis	Social and behavioural analysis on software repositories, e.g., [12].	semi-local

Choosing the best data storage strategy for different types of analyses is very important and challenging.

Challenge 5: Recovering from errors.

During the experiments, the machines in the cluster might crash and the MSR study tools used in the experiments might fail or throw exceptions. The MapReduce platform needs to catch failures and exceptions from both hardware and software during large-scale experiments. Handling and recovering errors is important when migrating MSR study tools to a MapReduce cluster.

4. CASE STUDIES

This section briefly presents the three case studies that we used to study how to address the challenges of migrating MSR tools to the MapReduce platform.

4.1 Subject systems and input data

We chose three representative MSR case studies and associated tools to counter potential bias. Prior research identifies eight major types of MSR analyses [25], as shown in Table 1. Techniques across these types require time-consuming processing and must cope with growing input data. We select three MSR tools that cover six out of the eight types of MSR analyses. Section 6 discusses the applicability of MapReduce to the two types of MSR analyses that are not covered by our case studies (i.e., visualization and social network analysis). We summarize below our case study tools.

J-REX. CVS repositories [3] contain the historical snapshots of every file in a software project, with a log of every change during the history of the software project. J-REX, similar to C-REX [22], processes CVS repositories to:

- Extract information (e.g., author name and change message) from each CVS transaction.
- Transform source code into an XML representation.
- Abstract source code changes from the line level ("line 1 has changed") to the program entity level ("function f1 no longer calls function f2") .
- Calculate software metrics, e.g., #LOC.

Table 2: Overview of the three subject tools.

	J-REX	CC-Finder	JACK
Programming Language	Java	Python	Perl
Source code	available	not available	available
Input data	Eclipse, Datatools	FreeBSD	Log files No. 1 & 2
Input data type	CVS repository	source code	execution log

Table 3: Characteristics of the input data.

	Data Size	Data Type	# Files
Eclipse	$10.4GB$	CVS repository	189,156
Datatools	$227MB$	CVS repository	10,629
FreeBSD	$5.1GB$	source code	317,740
Log files No.1	$9.9GB$	execution log	54
Log files No.2	$2.1GB$	execution log	54

J-REX performs 4 types of MSR analyses, i.e., Metadata analysis, Static source code analysis, Source code differencing and Software metrics [25].

CC-Finder. CC-Finder is a token-based clone detection tool [26] designed to extract code clones from systems developed in several programming languages (e.g., C++, and C). CC-Finder belongs to the clone detection analysis type.

JACK. JACK is a log analyzer that uses data mining techniques to process system execution logs, and automatically identify problems in load tests [24]. JACK performs the Metadata analysis and Data Mining MSR studies.

Source code of J-REX and JACK was available to us.

4.2 Experimental environment

To perform our evaluation, we require input data, a cluster of machines and a MapReduce implementation.

We use the CVS repository archives of Eclipse, a widely used Java IDE, and Datatools, a data management platform, as J-REX's input data. We downloaded the latest version of these archives on September 15, 2009. FreeBSD is an open source operating system. We use the source code distribution of FreeBSD version 7.1 as the input data for CC-Finder. Finally, two groups of execution log files are used as input data of JACK [24]. Tables 2 and 3 give an overview of the three software engineering tools and their input data.

Our experiments are performed on two clusters: 18 machines of a student lab and 10 machines of a scientific cluster called SHARCNET [7]. Table 4 shows the configuration of the two clusters. From previous research [35], we also have experience using a cluster in a research lab.

We choose Hadoop [39] as our MapReduce implementation. Hadoop is an open-source implementation of MapReduce supported by Yahoo! and widely used in industry. Hadoop not only implements the MapReduce model, but also provides a distributed file system, called the Hadoop Distributed File System (HDFS). Hadoop supplies Java interfaces to implement MapReduce operations and to control the HDFS. Another advantage for users is that Hadoop by default comes with libraries of basic and widely used "Map" and "Reduce" implementations, for example to break down files into lines. With these libraries, users occasionally do not have to write new code to use MapReduce.

4.3 Performance

To illustrate the scalability improvements of MapReduce for MSR analyses, we briefly discuss the performance obtained using MapReduce in our experiments, compared to the performance on a single machine without MapReduce. We repeated each experiment three times, and always report the median value of our results. A more detailed analysis of the performance gains of MapReduce for J-REX can be found in previous work [35]. Table 5 shows the best performance for each tool with the Eclipse CVS repository, FreeBSD source code and the 10GB system execution log files as input data respectively. For CC-Finder, we cannot perform code clone detection in the FreeBSD source code on one machine because of memory limitations. From the table, we can see that on a cluster of 10 machines (SHARCNET), the running time of J-REX and JACK is reduced by a factor 9 and 6 respectively. For CC-Finder, the running time is only 59 hours. Livieri *et al.* [28] claim that using CC-Finder to detect code clones in the FreeBSD source code requires 40 days. Although the experiments are performed on different hardware environments, the huge difference of running time gives an idea of the scalability of MapReduce.

Table 4: Configuration of MapReduce clusters.

	Student Lab	SHARCNET
# Machines	18	10
CPU	Intel Q6600 (2.4GHz)	$8 \times Xeon(3.0GHz)$
Memory	$3GB$	$8GB$
Network	Gigabit	Gigabit
OS	Ubuntu 8.04	CentOS 5.2
Disk size	$10GB$	$64GB$

Table 5: Best results for the migrated MSR tools.

Tool name	Input data	One machine	MapReduce version	Cluster
J-REX	**Eclipse**	$755min$	$80min$	SHARCNET
CC-Finder	**FreeBSD**	–	$59hours$	student lab
JACK	**Log file No.1**	$580min$	$98min$	SHARCNET

5. MIGRATION EXPERIENCES

While the previous section confirms that MapReduce can effectively scale several types of MSR analyses, it took us several attempts and experiments to achieve such performance results. In this section we distill our experience such that others would benefit from them. For each challenge from Section 3, we discuss our findings and provide advice based on our experience. We also compare our findings relative to common guidelines provided by the web field.

Challenge 1: Migrating MSR tools to a divide-and-conquer programming model.

We used the following strategies to map the MSR tools to a divide-and-conquer programming model.

J-REX. Similar to the original J-REX, the history of every single file is processed in isolation. Every input key/value pair contains the raw data of one file in the CVS repository. The Mappers pass the key/value pairs as *"file name/version number of the file"* to Reducers. Reducers perform computations to analyze the evolutionary information of all the revisions of a particular file. For example, if file "a.java" has three revisions, the mapping phase gets file names and revision numbers as input, and generates every revision number of the file, such as *"a.java/a_0.java"*. The Reducer generates *"a.java/evolutionary information of a.java"*. The full implementation details are discussed in [35].

CC-Finder. Our MapReduce implementation adopts the same computation model as D-CCFinder [28], which consists of the following steps:

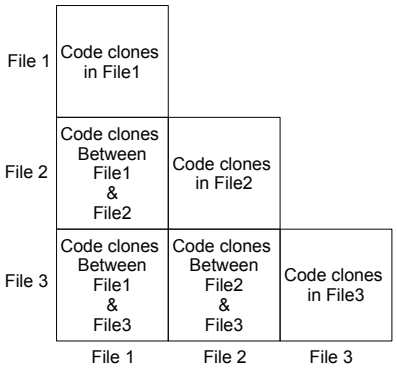

Figure 1: Example of the typical computational model of clone detection techniques.

1. Dividing source code into a number N of file groups.
2. Combining every two file groups together, resulting into $N \times (N+1)/2$ *"file group pair id/file names in both file groups"* pairs, which are sent to Mappers.
3. Mappers send the pair to a Reducer.
4. The Reducer invokes CC-Finder on a particular pair to run the clone analysis.

Figure 1 shows an example of the computational model for detecting code clones in 3 files. From the figure, we can see that every file needs to be compared to every other file and to itself, resulting into 6 pairs.

JACK. JACK detects system problems by analyzing log files. The Mapper receives every file name as input key/-value pair, and passes *"file name, file name"* to the Reducer. Passing only the file name instead of the file content avoids I/O overhead. Reducers receive the file name and invoke JACK to analyze the file.

In the global analyses, we have to put required data into Reducer. For local analyses, such as evolution of SLOC and JACK, we can use both Mapper and Reducer. Semi-local analyses can follow the migration strategy of either global or local analyses. Since the outputs of the different JACK invocations do not need to be aggregated, we only need one MapReduce phase instead of both Mappers and Reducers. We put all JACK functionality in Reducers, but could just as well have put it in the Mappers. Similar migration strategies are found in examples of MapReduce strategies such as "Distributed Grep" [15].

Notable Findings.

We summarize below our main observations.

1. **Locality of analysis.** A majority of MapReduce uses in the web field are local in nature, while for our case study we find that our three tools cover three levels of locality. The JACK tool performs local analysis, because the analysis of a file does not depend on other files. CC-Finder performs global analysis because every source code file must be compared to all the input source code files. J-REX performs semi-local analysis, because it compares consecutive revisions of every source code file. In another perspective, both J-REX and JACK have the algorithmic complexity of n, which is the number of input files; while CC-Finder has the algorithmic complexity of n^2. Yet, all tools show good performance after being MapReduced. For CC-Finder, we adopted the computation model proposed by [28] using the services provided by the MapReduce platform instead of spending considerable time implementing the platform for such a computation model. For J-REX, we found that for each analysis we needed a subset of the data (i.e., all consecutive revisions of a particular file), hence we had to ensure that all the data is mapped to the same machine in the cluster.
2. **Availability of source code.** When no source code was available, we used a program wrapper, which creates a process to call executable programs. When the source code was available, we sometimes had to use a program wrapper to invoke the tool because the tool and the MapReduce implementation used different programming languages (e.g., JACK is written in Perl while developers need to use Java on Hadoop). When the tool's source code was available and written in Java, e.g., for J-REX, the source code of the tool was modified to migrate to MapReduce.

Migrating local and semi-local analyses is much simpler than migrating global analyses. Little design effort is required for migrating J-REX and JACK. CC-Finder, as a global analysis, required more design effort than the other tools. We implemented 300 to 500 lines of Java code to migrate each tool.

Challenge 2: Locating a suitable cluster

In previous research [35], we used a four-machine MapReduce cluster in our research lab. In this paper, we used a cluster in a student lab and a cluster in SHARCNET. We document below our experiences using these three types of clusters.

Research lab. The heterogeneous nature of research labs complicates the deployment of MapReduce implementations such as Hadoop. These implementations require common configuration choices on every machine, such as a common user name and installation location. In an effort to reduce the complexity of deployments in research labs, we explored the use of virtual machines instead of the actual machines. The virtual machines unify the operating system, user name and installation location. However, virtual machines introduce additional overhead especially for I/O intensive analysis, while for CPU intensive analysis the overhead turned out to be minimal.

Student lab. The limited and unstable nature of storage in the student lab limited the use of Hadoop. All too often student labs provide too limited disk space for analysis and machines are typically configured to erase all space when booting up. The limited storage space prevented us from running experiments that performed global or semi-local analysis.

SHARCNET. While SHARCNET (and other scientific computing clusters) provide the desired disk space and homogeneous configuration, we were not able to use the main clusters of SHARCNET. Most scientific clusters make use of specialized schedulers to ensure fair sharing of the cluster, which do not support Hadoop. Fortunately, the SHARCNET operators gave us special access to a small testing cluster without scheduling requirements.

Notable Findings.

Heterogeneous infrastructures are not frequently used in the web field. Hence, the support provided by MapReduce implementations, like Hadoop, for such infrastructures is limited. In the research community, heterogeneous infrastructures are the norm rather than the exception. We hope that future versions of Hadoop will provide better support.

For now, we have explored the use of virtual machines on heterogenous infrastructures to provide a homogeneous cluster. The virtual machine solution works well for non-I/O intensive analysis and as a *playground* for analysis and debugging before deployment on larger clusters. We have used such a *virtual playground* to verify our MapReduce migration before deploying on expensive commercial Hadoop clusters, such as the Amazon EC2 Hadoop images [1].

While scientific clusters provide an ideal homogeneous infrastructure, their schedulers have yet to adapt to MapReduce's model. Researchers should work closer with the administration teams of scientific clusters such that MapReduce-friendly schedulers are adopted by these clusters.

Challenge 3: Optimization of MapReduce

We now discuss our observations regarding the optimization of MapReduce processing.

1) Static breakdown of analysis.

We explored the use of fine-grained (most often used in the web field) and coarse-grained breakdown in our migration of the different tools. For example, for the CC-Finder tool we started to read files from the input source code repository and record the size of every file until the total file size reached a certain threshold. The fine-grained breakdown processed 200MB of files per part while the coarse-grained breakdown processed 1GB of data per part (the CC-Finder version we had did not support more than 1GB of data). For J-REX, we explored the use of single files and sub-folders for breakdown granularity. In these experiments, we found that coarse-grained breakdown is two to three times faster than fine-grained breakdown because the processing time of each fine-grained unit has a large portion wasted on communication overhead. This finding confirms common knowledge in distributed computing.

2) Dynamic breakdown of processing.

We studied the impact of the dynamic breakdown of processing on performance by varying the number of processing tasks in Hadoop. We experimented with J-REX using the Datatools CVS repository and JACK using the Log files No.2, on 10 machines in SHARCNET. We set the number of Hadoop tasks to 10 (the number of machines) and recorded the running time of every machine in the cluster. In the violin plots of Figure 2, the top value corresponds to the maximum machine running time across all the machines, which determines the running time of the whole MapReduce process. The taller the grey box in the violin plot, the less balanced the workload of machines.

We then increased the number of Hadoop tasks to 100 for J-REX and 54 (the number of files, see Table 3) for JACK and compared the findings for the increased Hadoop task count to the performance of J-REX and JACK with just 10 Hadoop tasks. The plots in Figure 2 show that the running time of every machine after increasing the number of Hadoop tasks is more balanced than before (lower grey boxes in the violin plot). However, running JACK with more Hadoop tasks is faster than with fewer Hadoop tasks, while running J-REX with more Hadoop tasks is slower.

This contradictory result is caused by the different types of input data in the two software systems. The input data of J-REX is a CVS repository [3]. CVS repositories store the history of each file in a separate file, leading to a large number of input files. As shown in Table 2, JACK only has a few dozen files as input. The granularity of input files is finer for J-REX than for JACK. Increasing the number of

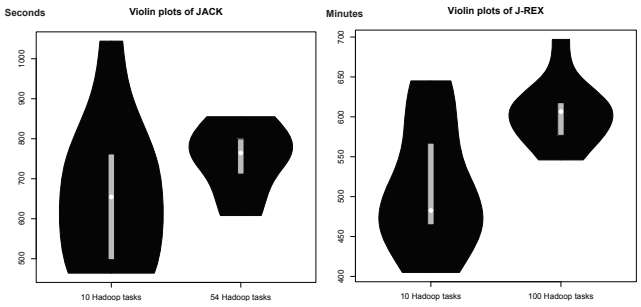

Figure 2: Violin plots of machine running-time for JACK and J-REX.

Hadoop tasks, yields a more balanced workload for both J-REX and JACK. However, this also increases the overhead of the platform to control and monitor Hadoop tasks. As a result, the best number of Hadoop tasks for J-REX seems to be the number of machines, i.e., coarsest granularity. For JACK, the best number of Hadoop tasks seems to be the number of input files, i.e., the finest granularity.

3) Determining the optimal number of machines.

To determine the optimal number of machines in our case study, we varied the number of machines from 5 to 10 on J-REX for the Datatools CVS repository and on JACK for the No.1 Log files. The number of Hadoop tasks are 10 and 54 (optimal dynamic breakdown) for J-REX and JACK respectively. Figure 3 shows the corresponding running times. We notice that the performance of J-REX grows sub-linearly, while the performance of JACK plateaus. Closer analysis indicates that this is primarily due to two reasons:

1. **Platform overhead.** The platform overhead is the time that the MapReduce platform uses to control Hadoop tasks, while the analysis time is the actual execution time of Mappers and Reducers. In our experiments, we find that the platform overhead is around 13% of the total running time with 5 machines and 23% of the total running time with 10 machines. Adding machines into the cluster introduces additional overhead. However, as the platform overhead is dominated by the analysis time when doing large-scale analysis, MapReduce performs better with larger scale analyses.
2. **Unbalanced workload.** An unbalanced workload causes machines to be idle. For example, a machine that is assigned much heavier work than others increases the total running time, as the whole MapReduce run will have to wait for that machine. In our experiments, unbalanced workload is the main reason for the un-optimal of JACK. In Figure 3, JACK does not improve its performance when moving from 6 machines to 10 machines. We checked the system logs of the MapReduce platform and found that one of the Hadoop tasks with the largest input log file took much longer than the other Hadoop tasks, which had to wait for that one Hadoop task to finish.

As a distributed platform, MapReduce requires transferring data over the network. Accessing a large amount of data also requires a large amount of I/O. Intuitively, I/O might be another possible source of the overhead. We observed the output of *vmstat* on every machine in the cluster and found that the percentage of CPU time spent on I/O is less than 1% on average, which means that in our experiment I/O was not a bottleneck.

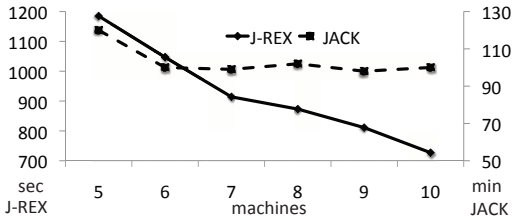

Figure 3: Running time trends of J-REX and JACK with 5 to 10 machines.

Notable Findings.

The web field often uses MapReduce to perform local analysis, with each broken-down part requiring substantial processing. In contrast, based on our case studies we note that many software engineering tasks (e.g., parsing a single file) require analyses that vary in locality. On the one hand, we would suggest researchers to analyze files in groups instead of individually in order to reduce platform overhead. However, the grouping of files might cause imbalance in the running time of Hadoop tasks, with some parts requiring more processing time than others. This in turn reduces the parallelism of the platform. In short, we can conclude that large-scale analysis on balanced input data benefits more from more machines in the cluster than small-scale analysis with unbalanced input data.

Our studies indicate that the recommended parameter configurations for using Hadoop on web data do not work well for all types MSR studies. For web data, it is recommended that the number of "Map" procedures is set to a value in between 10 to 100 $\times$ m, and that the number of "Reduce" procedures is set to 0.95 or 1.75 $\times$ m $\times$ n, with n being the number of machines and m being the number of processes that can run simultaneously on one machine, which is typically the number of cores of the machines [39].

This recommendation works well for web analysis, which is typically fine-grained. Fine-grained MSR tools like J-REX, which have a large number of input key/value pairs, can still adopt these recommendations. Coarse-grained MSR tools like JACK, which have a small number of input key/value pairs, should not adopt these recommendations. Instead, such tools should set the number of "Reduce" procedures to be the same as the number of input key/value pairs, i.e., the number of input files.

Challenge 4: Managing data during analysis

We used both distributed and local file systems.

1. **Distributed file system.** Hadoop offers a distributed file system (HDFS) to exchange data between different machines of a cluster. Such file systems are optimized for reading and perform poorly for writing data [39]. With many MSR tools generating a large number of intermediate files, the overhead of using HDFS is substantial. For example, if J-REX were to use HDFS when analyzing Eclipse, J-REX would require almost 190,000 writes to HDFS (a major slowdown). Therefore, we avoided the use of HDFS whenever possible, opting instead for the local file system. In the special case where no source code is available for an MSR tool, it might not even be possible to use HDFS, as accessing HDFS data requires using special APIs.
2. **Local file system.** In our experiments, we find that using every machine's local file system provides the most optimal solution of storing intermediate and output data. For example, CC-Finder and the log analyzer both output results to files, which we store in local file system. Since the output files are spread on different machines, we have to retrieve the results after the MapReduce run is completed. However, we have to take the risk of losing output data and re-performing the analysis when a machine crashes.

Notable Findings.

HDFS is the default data storage of Hadoop for the web analyses, but was not designed for fast data writing, which is necessary in saving MSR analyses result data. From our experience, we recommend: 1) the use of the local file system if the result data consists of a large amount of files; and 2) the use of HDFS if the result data is small in size.

Challenge 5: Error recovery

Our experiments evaluated the error recovery of Hadoop.

1. **Environment failure.** To examine the error recovery of Hadoop, we performed an experiment with J-REX and the Datatools CVS repository on 10 machines. First, we killed MapReduce processes and restarted them after 1 minute. We gradually increased the number of killed processes starting from 1 until the whole MapReduce job failed. Second, we did the same thing as the first step, but without restarting the processes. Our experimental results show that MapReduce jobs process well with up to 4 out of 10 machines killed. However, the running time increases from 12 min to 22 min. If we restore the working processes, the Hadoop job can finish successfully with up to half of the machines down at the same time.
2. **Tool error.** The strategy of addressing MSR tool errors depends on the implementation of the "Map" and "Reduce" procedures. If the MapReduce platform catches an exception, the platform will automatically re-start the Mapper or Reducer. According to our experience, if a program wrapper is used in the MapReduce algorithm, the wrapper needs to take the output of the MSR study tool, determine the running status, and throw an exception to the MapReduce platform to exploit MapReduce's tool error recovery. Alternatively, the wrapper can restart the analysis without throwing the exception to the MapReduce platform. In both cases, tool error can be caught and recovered.

Notable Findings.

We found that Hadoop's error recovery mechanism enabled us to have *agile clusters* with machines joining and leaving the cluster based on need. In particular, in our research lab students can join and leave a cluster based on their location and their current needs for the machine.

Because of Hadoop, an MSR tool might be executed millions of times. Hence, better reporting is needed by MSR tools such that any failure can be spotted easily within the millions of executions. We are currently exploring the use of techniques to detect anomalies in load tests (e.g., [24]) for detecting possible failures of the execution of an MSR tool.

6. APPLICABILITY

This section discusses the applicability of MapReduce to all the eight types of MSR analyses presented in Section 4. For each type, we present possible migration strategies. These strategies basically all depend on whether or not an analysis is local. We summarize in Table 6 the main challenges of migration, ease of migration and existence of prior research about scaling the analysis.

Table 6: Applicability of performing MSR analysis using the MapReduce platform.

Name	Main Challenge	Ease of migrating	Prior research
Metadata analysis	Challenge 3	easy	no
Static source code analysis	Challenge 1 & 3	easy or medium	no
Source code differencing and analysis	Challenge 3	easy	no
Software metrics	Challenge 3	easy or medium	no
Visualization	Challenge 1	hard	no
Clone-detection methods	Challenge 1	hard	yes, [28]
Data Mining	Challenge 1	hard	yes, [5]
Social network analysis	Challenge 1	medium	yes, [9]

Metadata analysis. In metadata analysis, data can just be broken down by the type of the metadata. For example, bug repository analysis can be broken down to analyzing individual bug reports.

Static source code analysis. Local static analyses can be migrated by breaking down the source code into several local parts and using a program wrapper to invoke the existing tools. If the static analysis process is non-local, the process of every source code file will consist of two steps: 1) collecting the required data in the other source code files; 2) performing analysis on the file and its collected data.

Source code differencing and analysis. The process can be broken down by files or by consecutive revisions. J-REX performs source code differencing.

Software metrics. The MapReduce strategies can be designed based on the types of software metrics. Our example of studying the evolution of #LOC of a software project in Section 3 is an example of a software metric.

Visualization. The visualization techniques that we consider consist of a regular MSR technique, followed by the generation of a visualization.

Clone-detection methods. Clone-detection techniques are non-local. This is the reason why they are hard to migrate to MapReduce. Livieri *et al.* [28] proposes an approach to map clone-detection to divide-and-conquer, which we adopted in our case study as MapReduce strategy.

Data Mining. Many Data Mining techniques require the entire data to build a model or to retrieve information, which makes Data Mining techniques hard to migrate to MapReduce. However, research has been performed to address the challenges of Data Mining algorithms to MapReduce. As such, some open source libraries are available for running Data Mining algorithms on Hadoop [5].

Social network analysis. Social networks can be analyzed as a graph with nodes and edges. Some of the analyses of the entire graph can be broken down to analyses of individual nodes or edges. X-RIME [9] is a Hadoop library for social network analysis.

Based on the examination of the 8 types of MSR analyses, most analyses are able to migrate to MapReduce, despite some challenges. Moreover, previous research (e.g., [5,9,28]) has addressed migrating some of the challenging analyses.

7. THREATS TO VALIDITY

We discuss the threats to validity for our findings.

Generalizability. We chose to scale three MSR tools. Although we chose tools across different types of MSR studies and using different subject systems to avoid potential bias of our studies to any special MSR study, our results may not generalize to other MSR studies. However, our case studies provide promising findings and we encourage other researchers to explore MapReducing their tools. Section 6 provides a brief discussion of generalization across other MSR studies.

Shared hardware environment. The scientific computing environment we used is a shared cluster. The usage of other users on the cluster may have impacted our case study results, which would threaten our findings. To counter this threat, we tried to use the cluster when it was idle, we repeated each experiment three times, and we reported the median value of the results.

Subjectivity bias. Some findings in our research can include subjectivity bias. For example, one of the MSR tools in our experiment was developed by the author of this paper, while the other two are not. Using our own tools for experimentation may cause subjectivity bias. However, in practice one will typically only alter the source code of tools that they know well. More case studies on other MSR tools are needed to verify our findings.

8. CONCLUSION

Automated software engineering tools continue to play an important role in the analysis of large data sets using sophisticated algorithms. In an effort to scale such tools, developers often opt for ad hoc, one-off solutions that are costly to develop and maintain. In this paper, we demonstrate that standard large-scale data processing platforms, like MapReduce, could be used to effectively and efficiently scale MSR tools, despite several challenges. We document our experiences such that others would benefit from them. We find that while MapReduce provides an efficient platform, we must follow different guidelines when configuring MapReduce runs instead of following the standard web field guidelines. In particular, software engineering analyses are often not local and software engineering analyses require different configuration than to web analyses to achieve optimal performance. We hope that our experiences will help others exploring the use of large-scale data analysis platforms to scale automated software engineering tools, instead of developing their own solutions.

9. REFERENCES

[1] Amazon EC2. https://aws.amazon.com/ec2/.
[2] checkmycode. http://www.checkmycode.org/.
[3] CVS. http://www.cvshome.org/.
[4] Debian counting. http://libresoft.es/debian-counting/.
[5] MAHOUT. http://lucene.apache.org/mahout/.
[6] Self-service, prorated super computing fun! http://open.blogs.nytimes.com/2007/11/01/.
[7] SHARCNET. https://www.sharcnet.ca.
[8] Vertica home page. http://www.vertica.com.
[9] X-RIME home page. http://xrime.sourceforge.net/.
[10] A. Acharya, G. Edjlali, and J. Saltz. The utility of exploiting idle workstations for parallel computation. *SIGMETRICS Perform. Eval. Rev.*, 25(1):225–234, 1997.
[11] S. Bajracharya, J. Ossher, and C. Lopes. Sourcerer: An internet-scale software repository. In *SUITE '09: Proceedings of the 2009 ICSE Workshop on*

Search-Driven Development-Users, Infrastructure, Tools and Evaluation, pages 1–4, Washington, DC, USA, 2009. IEEE Computer Society.

[12] O. Baysal and A. J. Malton. Correlating social interactions to release history during software evolution. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 7, Washington, DC, USA, 2007. IEEE Computer Society.

[13] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Extracting structural information from bug reports. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 27–30, New York, NY, USA, 2008. ACM.

[14] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey. Facilitating software evolution research with kenyon. In *ESEC/FSE '05: Proceedings of the 10th European Software Engineering Conference*, 2005.

[15] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51, 2008.

[16] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. Technical report, MS-TR-2007-58, Microsoft, May 2007.

[17] J. M. Gonzalez-Barahona, G. Robles, M. Michlmayr, J. J. Amor, and D. M. German. Macro-level software evolution: a case study of a large software compilation. *Empirical Softw. Engg.*, 14(3):262–285, 2009.

[18] C. Görg and P. Weißgerber. Error detection by refactoring reconstruction. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[19] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message passing interface*. MIT Press, 1999.

[20] M. Harman. The current state and future of search based software engineering. In *FOSE '07: 2007 Future of Software Engineering*, pages 342–357, Washington, DC, USA, 2007. IEEE Computer Society.

[21] M. Harman and B. F. Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833 – 839, 2001.

[22] A. E. Hassan. *Mining software repositories to assist developers and support managers*. PhD thesis, University of Waterloo, 2005.

[23] A. E. Hassan. The road ahead for mining software repositories. In *FoSM: Frontiers of Software Maintenance*, pages 48–57, October 2008.

[24] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automatic identification of load testing problems. In *ICSM '08: Proceedings of 24th IEEE International Conference on Software Maintenance*, pages 307–316, Beijing, China, 2008. IEEE.

[25] H. Kagdi, M. L. Collard, and J. I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint. Evol.*, 19(2):77–131, 2007.

[26] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[27] C. Kirsopp, M. J. Shepperd, and J. Hart. Search heuristics, case-based reasoning and software project effort prediction. In *GECCO '02: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1367–1374, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[28] S. Livieri, Y. Higo, M. Matushita, and K. Inoue. Very-Large Scale Code Clone Analysis and Visualization of Open Source Programs Using Distributed CCFinder: D-CCFinder. In *ICSE '07: Proceedings of the 29th International conference on Software Engineering*, 2007.

[29] W. Maalej and H.-J. Happel. From work to word: How do software developers describe their work? In *MSR '09: Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 121–130, Washington, DC, USA, 2009. IEEE Computer Society.

[30] A. Mockus. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In *MSR '09: Proceedings of 6th IEEE International Working Conference on Mining Software Repositories*, pages 11–20, 2009.

[31] L. Moonen. Generating robust parsers using island grammars. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 13, Washington, DC, USA, 2001. IEEE Computer Society.

[32] A. P. Nikora and J. C. Munson. Understanding the nature of software evolution. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 83, Washington, DC, USA, 2003. IEEE Computer Society.

[33] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, New York, NY, USA, 2008. ACM.

[34] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, 2009.

[35] W. Shang, Z. M. Jiang, B. Adams, and A. E. Hassan. Mapreduce as a general framework to support research in mining software repositories (MSR). In *MSR '09: Proceedings of 6th IEEE International Working Conference on Mining Software Repositories*, pages 21–30, 2009.

[36] Y. Shin, R. Bell, T. Ostrand, and E. Weyuker. Does calling structure information improve the accuracy of fault prediction? In *MSR '09: Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 61–70, Washington, DC, USA, 2009. IEEE Computer Society.

[37] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. Mapreduce and parallel dbmss: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.

[38] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, 2004.

[39] T. White. *Hadoop: The Definitive Guide*. Oreilly & Associates Inc, 2009.