Enabling Large-Scale Mining Software Repositories (MSR) Studies Using Web-Scale Platforms

by

Weiyi Shang

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
May 2010

Abstract

The Mining Software Repositories (MSR) field analyzes software data to uncover knowledge and assist software developments. Software projects and products continue to grow in size and complexity. In-depth analysis of these large systems and their evolution is needed to better understand the characteristics of such large-scale systems and projects. However, classical software analysis platforms (e.g., Prolog-like, SQL-like, or specialized programming scripts) face many challenges when performing large-scale MSR studies. Such software platforms rarely scale easily out of the box. Instead, they often require analysis-specific one-time ad hoc scaling tricks and designs that are not reusable for other types of analysis and that are costly to maintain. We believe that the web community has faced many of the scaling challenges facing the software engineering community, as they cope with the enormous growth of the web data. In this thesis, we report on our experience in using MapReduce and Pig, two web-scale platforms, to perform large MSR studies. Through our case studies, we carefully demonstrate the benefits and challenges of using web platforms to prepare (i.e., Extract, Transform, and Load, ETL) software data for further analysis. The results of our studies show that: 1) web-scale platforms provide an effective and efficient platform for large-scale MSR studies; 2) many of the web community's guidelines for using web-scale platforms must be modified to achieve the optimal performance for large-scale MSR studies. This thesis will help other software engineering researchers who want to scale their studies.

Acknowledgments

This thesis would not have been possible without the continuous support of my parents who always support me and give me the will to succeed.

I would like to thank my supervisor Dr. Ahmed E. Hassan for his support and advice. A special thank you to Dr. Bram Adams for his fruitful suggestions throughout my research during the last one and a half years.

In addition, I appreciate the valuable feedback provided by two of my thesis readers: Dr. Patrick Martin and Dr. Ahmad Afsahi.

I appreciate the lively and engaging discussions with all the members of SAIL. In particular, I would like to thank Zhen Ming Jiang, for all his help and encouragement.

I would like to thank SHARCNET and Ben Hall for providing the experimental environment used in this thesis.

Finally, I thank all my friends who patiently put up with me while I was working on my thesis.

Related Publications

The following is a list of our publications that are on the topic of enabling large-scale Mining Software Repositories (MSR) studies using web-scale platforms:

- MapReduce as a General Framework to Support Research in Mining Software Repositories (MSR), Weiyi Shang, Zhen Ming Jiang, Bram Adams and Ahmed E. Hassan, MSR '09: Proceedings of the 6th IEEE Working conference on Mining Software Repositories (MSR), 2009, Vancouver, Canada, 2009. This work is described in Chapter 3.

- An Experience Report on Scaling Tools for Mining Software Repositories Using MapReduce, Weiyi Shang, Bram Adams and Ahmed E. Hassan, submitted to ASE '10: The 25th IEEE/ACM International Conference on Automated Software Engineering. This work is presented in Chapter 4.

- Enabling Large Scale Software Studies Using Web-Scale Platforms: An Experience Report, Weiyi Shang, Bram Adams and Ahmed E. Hassan, submitted to SCAM '10: Tenth IEEE International Working Conference on Source Code Analysis and Manipulation. This work is discussed in Chapter 5.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software projects and systems continue to grow in size and complexity. The first version of the Linux kernel, which was released in 1994, consists of 176,250 lines of source code, while version *2.6.32* released in 2009 consists of 12,606,910 lines of source code. In fifteen years, the size of Linux kernel has grown more than 70 folds. Similarly, Gonzalez-Barahona *et al.* find that the size of the Debian Linux distribution doubles approximately every two years [30, 64]. Moreover, recent work by Mockus shows that a universal repository of the version history of software systems contains TBs of data and that the process to collect such a repository is rather lengthy and complicated, taking over a year [57]. The size of the available code continues to grow and so do the challenges of amassing and analyzing such large code bases.

This explosive growth in the availability and size of software data has led to the formation of the Mining Software Repositories (MSR) field [37]. The MSR field recovers and studies data from a large number of software repositories, including source control repositories, bug repositories, archived communications, deployment logs, and code repositories. Until today many of such large-scale studies are performed on data prepared by classical platforms (e.g., Prolog-like, SQL-like, or specialized programming scripts). Such prepared data is then further analyzed using modeling tools like R [42], Weka [33], or other specially built tools. The concept of data preparation is often referred to as ETL: Extract, Transform, and Load [69].

As the size of software data increases, more complex platforms are needed to enable rapid and efficient data preparation (ETL). Software engineering studies typically try to scale up ETL by means of specialized one-off solutions that are not easy to reuse and that are costly to maintain. For example, to identify clones across the FreeBSD operation system (131,688k LOC), Livieri *et al.* developed their own distributed platform instead of reusing other platforms [53]. We believe that in many ways the ETL phase is not of interest to many researchers, instead they prefer to focus on their core competency, i.e., the analysis techniques, which analyze the prepared data.

Many of the challenges associated with data preparation in the software engineering community have already been faced by the web community. The web community has developed several platforms to enable the large-scale preparation and processing of large-scale data sets. MapReduce [24], Hadoop [70], Pig [60], Hive [5], Sawzall [62] and SCOPE [22] are examples of such platforms. Because of the similarity between MSR studies and web analyses, we firmly believe that our community can adopt many of these platforms to scale data preparation of MSR studies.

In this thesis, we explore the use of popular web-scale platforms for performing large-scale MSR studies. Through our case studies we carefully demonstrate the feasibility and experience of using MapReduce to scale the data preparation phase in MSR studies, and using Pig to improve the re-usability and maintainability of software-specific ETL operations.

This chapter consists of the following parts: Section 1.1 presents our research hypothesis. Section 1.2 gives an overview of the thesis. Section 1.3 briefly discusses the contributions of this thesis. Section 1.4 presents the organization of the thesis.

1.1 Research hypothesis

Prior research and our experience lead us to the formation of our research hypothesis. We believe that:

The need for large-scale Mining Software Repositories (MSR) studies continues to grow as the size and complexity of studied systems and analysis techniques increase. Existing approaches to scale MSR studies are not re-usable and are often costly to maintain. The need for scalable studies is very prominent in the MSR field. We believe the MSR field can benefit from web-scale platforms to overcome the limitation of current approaches.

The goal of this thesis is to empirically explore this hypothesis by performing large-scale MSR studies using web-scale platforms. In particular, we perform three different MSR studies on a MapReduce platform to validate the feasibility of scaling MSR studies and to generalize our experience of migrating MSR studies to such web-scale platforms. We then use the Pig platform to improve the re-usability and maintainability of scaling efforts for MSR studies in practice.

Moreover, we document our experiences and evaluate the guidelines from the web community when using such platforms to scale MSR studies. Our experiences show that the standard guidelines from the web community must be changed to avoid sub-optimal performance for MSR studies that use web-scale platforms. Our documented experiences and source code can assist other MSR researchers and practitioners in scaling other MSR studies.

1.2 Thesis overview

We now give an overview of the work presented in this thesis.

1.2.1 Chapter 2: Background

The Mining Software Repositories (MSR) field analyzes data in software repositories to uncover valuable information of software projects and assist software engineering practitioners [37].

The need for large-scale MSR studies continues to grow as the size and complexity of studied systems and analysis techniques increase, and more different kinds of repositories become available. MSR 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 re-usable and are often costly to maintain. The need for scalable studies is very prominent in the MSR field. Chapter 2 presents the two backgrounds of this thesis: 1) two trends related to the need of large-scale MSR studies; and 2) existing approaches to scale MSR studies.

Large-scale data analysis is performed everyday by companies such as Google and Yahoo!. The web community has developed several platforms to enable the large-scale analysis. These platforms are able to run on a distributed computing environment to gain computing power and increase data storage space. Data analysis algorithms can be migrated to these platforms without having to know about network programming details. This chapter presents several widely used data analysis platforms in the web community and uses examples to illustrate the data analysis processes using these platforms. The introduction and examples of these platforms show that they hold promise in enabling large-scale MSR studies.

1.2.2 Chapter 3: Feasibility Study of Scaling MSR Studies using MapReduce

We propose the use of MapReduce, a web-scale platform, to support research in MSR. As a feasibility study, we migrate an MSR tool to run on Hadoop, an open source implementation of MapReduce. Through a case study on the source control repositories of the Eclipse, BIRT and Datatools projects, we demonstrate that the migration effort to MapReduce is minimal and that the benefits are significant, as the running time of the migrated J-REX is only 30% to 40% the time of the original MSR tools.

1.2.3 Chapter 4: MapReduce as a General Platform for Scaling MSR studies

In this chapter, we propose to use MapReduce as a general platform to scale more types of software studies in MSR. We performed three representative case studies from the MSR field to demonstrate that the MapReduce platform could be used as a general platform to successfully scale MSR studies with minimal effort. We document our experience such that other researchers could benefit from them. We also note that many of the web community's guidelines for using the MapReduce platform need to be modified to better fit the characteristics of MSR problems, otherwise we risk gaining sub-optimal benefits from such a scalable platform.

1.2.4 Chapter 5: Large-Scale MSR Studies with Pig

We find some of the limitations of MapReduce as a platform for large-scale MSR studies from the case studies in Chapter 3 and 4. In particular, the use of the MapReduce platform requires in-depth knowledge of the data processing phases of MSR studies and requires additional effort of designing Map and Reduce strategies for every data processing phase. In this chapter, we report on our experience in using Pig to improve the re-usability of scaling MSR studies. Through three case studies we carefully demonstrate the use of Pig to prepare data in software repositories for further analysis.

1.3 Thesis contributions

In this thesis, we demonstrate that web-platforms can be used to effectively enable large-scale MSR studies. We document our experience and find that standard guidelines from the web community may lead to sub-optimal performance due to the different characteristics of MSR data and problems in comparison to the web field. Our experience can assist other software engineering researchers interested in scaling their tools and performing large-scale software studies.

In particular, our contributions are as follows:

1. We verified the feasibility and benefits of scaling MSR studies using the MapReduce and Pig platforms.

2. We documented our experience in scaling MSR studies using MapReduce and Pig, and provided code samples of program units of Pig, such that other researchers could benefit from our experience and code samples.

3. We also note that changes are needed to the web community's standard guidelines for the MapReduce platform when migrating MSR studies to web-scale platforms. These changes highlight the different characteristics of MSR studies and must be done to ensure that MSR researchers get the most benefits from such web-scale platform.

1.4 Thesis organization

The rest of this thesis is organized as follows: Chapter 2 presents the background of this thesis and introduces web platforms for data analysis. Chapter 3 presents a feasibility study of using MapReduce for scaling MSR studies. Chapter 4 presents our experience of using MapReduce as a general platform for scaling MSR studies. Chapter 5 shows using Pig to improve the re-usability and maintainability of scaling MSR studies in practice. Finally, Chapter 6 summarizes our work and presents our future work.

Chapter 2

Background

MSR researchers continue to perform large-scale software studies on ever larger amounts of data using ever more complex algorithms and techniques. However, few of the approaches address the challenges of performing large-scale MSR studies. This chapter gives a brief introduction and several examples of MSR studies and illustrates two trends of MSR studies, i.e., the larger amounts of data and the use of more complex algorithms. Moreover, we explore the existing approaches for large-scale MSR studies to motivate the need and benefits of using web-scale platforms to support such studies. We then present the web-scale platforms and show simple examples of using two of these platforms to perform MSR studies.

2.1 Mining Software Repositories (MSR)

Various types of data are generated during software engineering activities. Source code, bug report and system execution logs are examples of software data. Large amounts of different software data are archived and stored in software repositories, such as source control repositories and bug repositories [37].

The Mining Software Repositories (MSR) field analyzes the data in software repositories to uncover knowledge to assist software engineering researchers and practitioners [37]. For example, software developers are required to write text logs when they commit a change to source code. The text log stored in the source control repositories can assist other developers understanding the purpose of the change and the architecture of the software [39].

We here present three MSR studies that are mentioned in this thesis:

- **Software evolution.** Software systems are continuously evolving during the history of software projects [40]. Studying the evolution of a software system assists developers in maintaining and enhancing the system. Typically, an evolutionary study of software systems can be performed at one of five levels [40]:

    1. *System Level:* At system level, one number as a measurement of every snapshot of a software system is generated. Lehman *et al.* studied such measurements and generalized eight laws for software evolution [52].
    2. *Subsystem Level:* At subsystem level, the evolution study is performed on every subsystem of the software project to understand more details about the evolution of a software system. For example, Godfrey *et al.* study the evolution of lines of code of every subsystem of the Linux kernel [29].
    3. *File Level:* The evolution study at file level reports the evolution of every source file of a software project. For example, the evolution study would report evolutionary information such as file *"a.c"* is 5 lines less after the most recent change.

4. *Code Entity Level:* At code entity level, every snapshot of every source code entity, such as a function, is recorded. The evolution study performs differencing analysis to generate results such as "Class *A* is changed, function *foo* is added into Class *A*.".

5. *Abstract Syntax Tree (AST) level:* At this level, an AST is built for every snapshot of the source code. The evolution study is based on tracking the evolution of the AST. For example, the evolution study at AST level may report that an *if statement*'s condition is changed.

- **Code clone detection.** Code clones are identical or near identical segments of source code [65]. There are two contradicting views of clone detection. One view believes that code clones may increase the maintenance effort and introduce bugs [54]. The other view considers code clone as a pattern in software development [48]. Detecting code clones can assist in software maintenance and program comprehension.

- **System log analysis.** Large amounts of log data are generated during software system testing and execution. Analyzing system logs can assist software developers and maintainers in finding bugs and system problems that are difficult to reveal by other software debugging and testing approaches [45].

In next section, we present a typical data pipeline in MSR studies.

2.2 Data pipeline in MSR

Figure 2.1 shows the typical pipeline for MSR studies, which starts with data preparation. The figure shows that the process of data preparation consists of the following

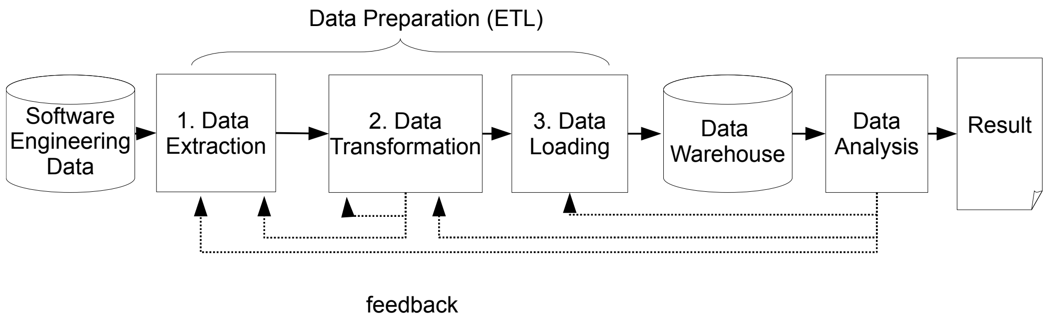

Figure 2.1: Pipeline of large-scale MSR studies.

three phases:

1. **Data Extraction.** Most data gathered during the software engineering process was not anticipated to be used for empirical studies. In order to extract actionable data, special tools are needed to process software repositories. Such tools are typically implemented in general-purpose programming languages. For example, bug repositories track the histories of bug reports and feature requests. Bettenburg *et al.* [19] built a tool called infoZilla that extracts structural data from bug reports into a relational database. Jiang *et al.* [44] developed an automated approach to extract data from system execution logs.

2. **Data Transformation.** After the raw data is extracted from software repositories and software archives, it typically needs to be abstracted and merged with other extracted data for further analysis. Such transformations can be implemented using general-purpose programming languages, Prolog-like languages, SQL-like languages, or Web technologies such as MapReduce [24].

   Maruyama *et al.* [56] developed an XML representation of Java source code. Robles *et al.* [63] use various software engineering data, such as source code, mailing list data and bug reports to merge different IDs of a single developers.

Emden *et al.* [68] used Grok to perform automatic code inspection for identifying *code smells* such as code duplication.

3. **Data Loading.** In this phase, the transformed data is converted into the format required by analysis tools and is loaded into various types of analysis environments, such as relational database, R and Weka, for further analysis. Data loading can even be as simple as writing transformed results to files that are the input of data analysis.

Data preparation (ETL) is a highly interactive process. For example, if the results of statistical analysis in the data analysis step look suspicious, researchers need to examine the output of the data extraction, transformation and loading phases, refine the phases, and re-prepare the data.

After the data preparation, various types of data analysis techniques can be performed on the prepared data. The data analysis tools generate the final results of the MSR studies.

2.3 Trends in MSR

In recent years, two major trends can be observed in the MSR field. The first trend is that the data analyzed by MSR studies 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.* [30] 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 [17, 57]. Estimation indicates that an entire year of processing is needed to amass such large source code [57]. This growth of data is not slowing down. For example, studies show that the Debian distribution is doubling in size approximately every two years [30, 64].

A second trend in the 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 [47], 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 [65], more complex techniques that require much more computing power and running time are needed to detect more types of code clones with higher precision.

2.4 Approaches for scaling 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. Other approaches, such as D-CCFinder [53], Kenyon [20] and SAGE [28], re-engineer the original non-distributed MSR study tools to enable them to run on a distributed environment. Distributed computing libraries, such as OpenMP [23], can assist in developing distributed MSR study tools. However, the above types of ad hoc solutions require additional programming effort or physical changes to programs. Moreover, few of the approaches are able to handle massive amounts of data, do error recovery and scale automatically.

Over the past 20 years, parallel database systems, such as Vertica [11], have been used to perform large-scale data analyses. Recently, work by Stonebraker *et al.* [67] 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 require extracting large amounts of data from software repositories and read the data sets without updating. Using parallel database system is not an optimal solution because database systems are hard to use with data extraction and ad hoc analyses. In addition, database systems are not designed for scan-centric workloads.

Search-based software engineering (SBSE) [35] 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.* [50] use various search algorithms to find an accurate cost estimate for software projects. Antoniol *et al.* [15] use search-based techniques to assist large-scale software maintenance projects. In addition to optimized performance, most search algorithms are naturally parallelizable to support even larger scale experiments [34]. However, SBSE only offers a set of general techniques to solve problems, and considerable application-specific customization is still needed to achieve significant speed-ups. Not all MSR analyses benefit from approximate solutions either.

Next section presents web-scale platforms such as MapReduce and Pig, which are promising to assist in enabling large-scale MSR studies.

2.5 Web-scale platforms

The web community has developed large-scale data analysis platforms over the years to support the data intensive processing in the web field. The web analyses have the following characteristics.

1. **Large input data sets.** Large amounts of data are processed by the web field. The input data may consist of TB-sized files.

2. **Scan-centric job.** Most of the processing only scans the entire input data (without updating) and generates relatively small output results.

3. **Fast evolving data.** The input data evolves fast and frequently. The data is mostly processed once and is thrown away.

Simple programming models or high-level programming languages are typically used on the web-scale platforms to minimize the programming effort. The web-scale platforms are typically leveraged by a distributed data storage technique such as Google File System (GFS) [27] and Hadoop Distributed File System (HDFS) [4].

The data preparation (ETL) in the MSR studies has similar characteristics as the analyses typically used in the web field. We believe that web-scale platforms can be adopted by the MSR community to enable large-scale MSR studies. This thesis focuses on two such platforms, which are introduced and illustrated below.

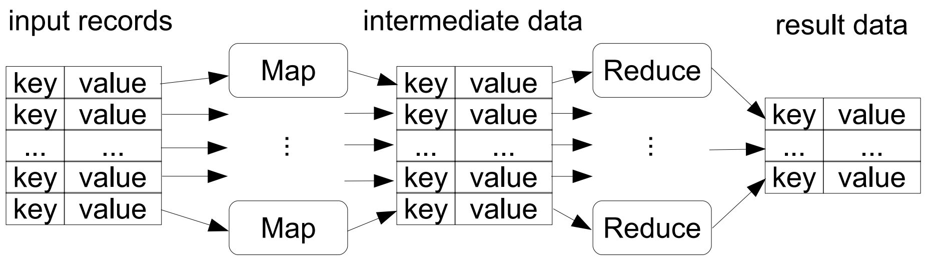

Figure 2.2: MapReduce programming model.

2.5.1 MapReduce

MapReduce is a web-scale platform for processing very large data sets [24]. The platform is proposed by Google and is used by Google on a daily basis to process large amounts of web data.

MapReduce enables a distributed divide-and-conquer programming model. Shown in Figure 2.2, the programming 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. MapReduce processes can be chained to implement analyses that consist of multiple steps.

An Example of MapReducing an MSR analysis. To illustrate how MapReduce can be used to support MSR studies, we consider performing a classical MSR study of the evolution of the total number of lines of code (#LOC) of a software project. The input data of this MSR study 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 group 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"*, for example, the Reducer will sum the values *"100"* and *"200"* and output *"1.0/300"*.

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

1. **a mature and proven platform.** MapReduce is widely used with great success by the web community 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 [9].

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 studies more affordable.

3. **a read-optimized platform.** MapReduce is designed to perform read-only data analyses. The optimization of read-only analyses can well support the scan-centric MSR studies.

Hadoop is an open-source implementation of MapReduce that is supported by

Yahoo and is 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), to store data. Hadoop supplies Java interfaces to implement MapReduce operations and to control the HDFS programmatically. 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, or to break down a directory into files. With these libraries, users occasionally do not have to write new code to use MapReduce.

2.5.2 Pig

Pig [60] is a platform designed for analyzing massive amounts of data built on top of Hadoop, an open-source implementation of MapReduce. Pig provides a high-level data processing language called Pig Latin [60]. Developers can use Pig Latin to develop programs that are automatically compiled to MapReduce programming model instead of designing "Map" and "Reduce" strategies.

To illustrate how Pig can be used for data preparation in MSR studies, we use it to measure the evolution of the total number of lines of code (#LOC) in the different snapshots of a source code repository. The corresponding source code in Pig Latin is shown in Figure 2.3. To better illustrate the Pig Latin script, all variables in our implementation use upper case, all Pig Latin key words use lower case, and the names of user defined functions use camel case.

In the source code shown in Figure 2.3, line 1 loads all data from a CVS repository into Pig storage, which is based on the Hadoop Distributed File System [4] (HDFS), as a (*file name, file content*) pair.

```
1 RAWDATA = load 'EclipseCvsData' using
      ExtPigStorage() as (filename:
    chararray, filecontent:chararray)
    ;
2 HISTORYLOG = foreach RAWDATA generate
      ExtractLog(filename, filecontent
    );
3 HISTORYVERSIONS = foreach HISTORYLOG
    generate ExtractVersions($0);
4 CODE = foreach HISTORYVERSIONS
    generate ExtractSourceCode($0);
5 LOC=foreach CODE generate GenLOC($0);
6 dump LOC;
```

Figure 2.3: Pig Latin script for measuring the evolution of the total number of lines of code (#LOC) in the different snapshots of a source code repository.

Line 2 extracts CVS log data of every source code file. Each of the program units in Pig, such as *ExtractLog* in line 2, is implemented as a Java Class with a method named *exec*. The Java source code of the *exec* method of the program unit *ExtractLog* is shown in Figure 2.4. In the Java source code shown in Figure 2.4, the parameter of method *exec* is a (*"CVS file name"*, *"CVS file content"*) tuple. Because the *rlog* tool that generates the historical log of CVS files needs a file as input, lines 7 to 10 write the file content to a temporary file. Line 11 generates the historical log by calling the method *extractRlog* that wraps the tool *rlog*. Lines 12 to 15 create and return a new (*"CVS file name"*, *"CVS historical log"*) tuple. The whole method contains less than 20 lines of code and uses an existing tool to complete the process.

In the remainder of the Pig Latin script in Figure 2.3, line 3 parses every source code file's log data in order to generate the historical version numbers of every source code file. After generating the version numbers, line 4 uses CVS commands and extracts source code snapshots of every file. Line 5 counts the LOC of each snapshot of every source code file and Line 6 outputs the result data.

```
public Tuple exec(Tuple input) throws IOException {
        if (input == null || input.size() == 0)
            return null;
        try{
            String name = (String) input.get(0);
           String content=(String) input.get(1);
            File file=new File(name);
             FileWriter fw=new FileWriter(file);
            fw.write(content);
            fw.close();
           String rlog=extractRlog(name);
           Tuple tname = DefaultTupleFactory.getInstance().newTuple();
           tname.append(name);
           tname.append(rlog);
         return tname;
        }catch(Exception e){
            throw WrappedIOException.wrap("Caught␣exception␣processing␣input␣row␣", e);
        }
}
```

Figure 2.4: Java source code of the *exec* method of the programming unit *"ExtractLog"* (generating source code history log).

We can see that the whole process of measuring the evolution of #LOC contains 4 program units: *"ExtractLog"*, *"ExtractVersions"*, *"ExtractSourceCode"*, and *"GenLOC"*, and a general data loading method *ExtPigStorage*.

Pig has the following characteristics that benefit large-scale MSR studies.

1. In the above example, Pig Latin combines a number of program units that provide different functionalities, which illustrates the modular design of Pig programs.

2. Pig compiles its source code to MapReduce, such that the program can scale to analyze massive amounts of data automatically.

3. Data preparation with the Pig platform is debuggable because Pig enables data previewing and sampling and all intermediate data is stored as variables in Pig.

2.5.3 Other web-scale platforms

In addition to MapReduce and Pig, we briefly reviewed some of the most notable web-scale platforms that are not focused in this thesis.

Similar to MapReduce and Pig, Microsoft develops a large-scale data processing platform called Dryad [43] and a corresponding high-level data processing language on top of Dryad named SCOPE [22]. Using Dryad, developers define a communication flow of the subroutines of the data processing tool and build a communication graph with the subroutines being vertices in the graph. The data processing tool can run in a distributed environment with Dryad [43]. In addition, using SCOPE, a scripting language on top of Dryad, the developers save programming effort in migrating data processing tools to the Dryad platform.

Sawzall [62] is a high-level scripting language proposed by Google to hide the details of MapReduce platform. Similar to Pig, the scripts of Sawzall are automatically compiled to programs following Google's MapReduce programming model.

As SQL is widely used in data analysis, an SQL-like language called Hive [5] is developed on top of Hadoop. Developers can write scripts similar to SQL and the scripts are automatically compiled to MapReduce code of Hadoop.

Even though Dryad, SCOPE and Sawzall are well designed, highly optimized and widely used in the web field, they are not open-sourced projects, such that we cannot use them in our studies. Hive provides the data analysis functionality following the data preparation (ETL), hence we do not focus on using Hive in this thesis. The use of Hive to scale software data analysis is in our future work.

2.6 Chapter summary

The Mining Software Repositories (MSR) field analyzes the data in software repositories to uncover knowledge and assist decision making process in software projects. Typical MSR process consists of a data preparation (ETL) phase and a data analysis phase on the prepared data by the data preparation phase. Ever larger data sets are processed by the MSR studies with more complex algorithms and techniques. Yet, few existing work scales the MSR studies easily and efficiently. Several web-scale platforms are developed to perform web analyses that are similar to MSR studies. We strongly believe these web-scale platforms can help in scaling MSR studies. In the next chapter, we use MapReduce as an example of a web-scale platform to evaluate the feasibility of scaling MSR studies using such platforms.

Chapter 3

Feasibility Study of Scaling MSR Studies using MapReduce

Introduced in Chapter 3, web-scale platforms are designed to process large amounts of data in the web field. These platforms are promising to scale MSR studies. In this chapter, we explore one of these platforms, called MapReduce [24], to evaluate the feasibility of scaling MSR studies with web-scale platforms. As a proof-of-concept, we migrate J-REX, an MSR study tool for studying software evolution, to run on the MapReduce platform. Through a case study on the source control repositories of the Eclipse, BIRT and Datatools projects, we show that J-REX is easy to migrate to MapReduce and that the benefits are significant. The running time of the migrated J-REX version is 30% to 40% of the running time of the original J-REX.

3.1 Requirements

Based on our experience, we seek four common requirements for large distributed platforms to support MSR research. We detail them as follows:

1. **Adaptability:** The original MSR tools should be easily migrated to MapReduce by MSR researchers without major re-engineering.

2. **Efficiency:** The adoption of the platform should drastically speed up the mining process.

3. **Scalability:** The platform should scale with the size of the input data as well as with the available computing power.

4. **Flexibility:** The platform should be able to run on various types of machines, from expensive servers to commodity PCs or even virtual machines.

This chapter presents and evaluates MapReduce as a possible web-scale platform that satisfies these four requirements.

3.2 Case study

This section presents our case study to evaluate using MapReduce, as an example of the web-scale platforms, to scale MSR experiments.

3.2.1 J-REX

To validate the promise of MapReduce for MSR research, we discuss our experience migrating an MSR study tool called J-REX to MapReduce. J-REX is used to study

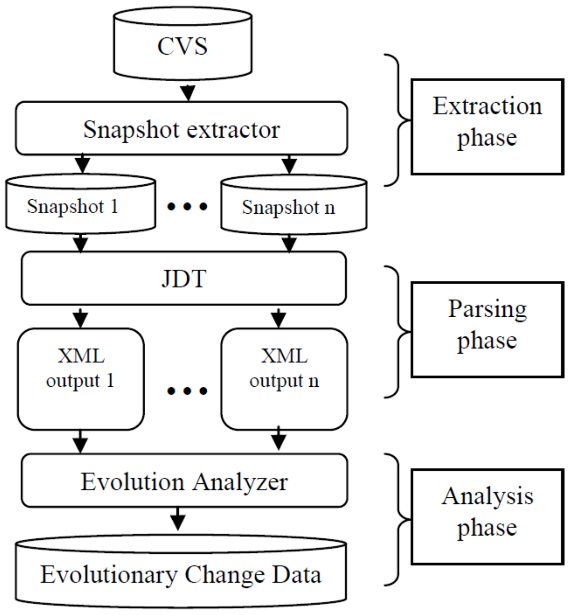

Figure 3.1: The Architecture of J-REX.

the evolution of source code of software systems developed in Java programming language, similar to C-REX [36]. J-REX performs software evolution study at code entity level, which is presented in Section 2.1.

As shown in Figure 3.1, the whole process of J-REX spans three phases. The first phase is the extraction phase, where J-REX extracts source code snapshots for each file from a CVS repository. In the second phase, i.e. the parsing phase, J-REX calls the Eclipse JDT parser for each file snapshot to parse the Java code into its abstract syntax tree [3], which is stored as an XML document. In the third phase, i.e. the analysis phase, J-REX compares the XML documents of consecutive file revisions to determine changed code units, and generates evolutionary change data in an XML format [39]. The evolutionary change data reports the evolution of a software system at the level of code entities such as methods and classes (for example, "Class *A* was changed to add a new method *B*"). J-REX has a typical MSR data pipeline corresponding to the data pipeline shown in Figure 2.1.

The J-REX runtime process requires a large amount of I/O operations that lead to performance bottlenecks. J-REX also requires a large amount of computing power for comparing XML trees. The I/O and computational characteristics of J-REX make it an ideal case study to study the performance benefits of the MapReduce platform.

3.2.2 MapReduce strategies for J-REX

This sub-section explains the design of migrating J-REX to MapReduce, which is referred as "MapReduce strategy" in this thesis. Intuitively, we need to compare the differences between adjacent revisions of a Java source code file. We could define the key/value pair output of the Mapper function as (D1, *a_0.java* and *a_1.java*), and the Reducer function output as (revision number, evolutionary information). The key D1 represents the differences between two versions, *a_0.java* and *a_1.java* represent the names of two files. Because of the way that we partition the data, each revision needs to be copied and transferred to more than one Mapper node. For example, $a_1.java$ needs to be transferred to both Mapper node with $a_0.java$ and Mapper node with $a_2.java$. This generates extra I/O overhead, and turns out to make the process much longer. The sub-optimal performance of this naive strategy shows the importance of designing a good strategy of MapReduce.

Therefore, we tried another basic MapReduce strategy, as shown in Figure 3.2. This strategy performs much better than our naive strategy. The key/value pair output of the Mapper function is defined as (file name, revision snapshot), whereas the key/value pair output of the Reducer function is (file name, evolutionary information for this file). For example, file *a.java* has 3 revisions. The mapping phase gets the

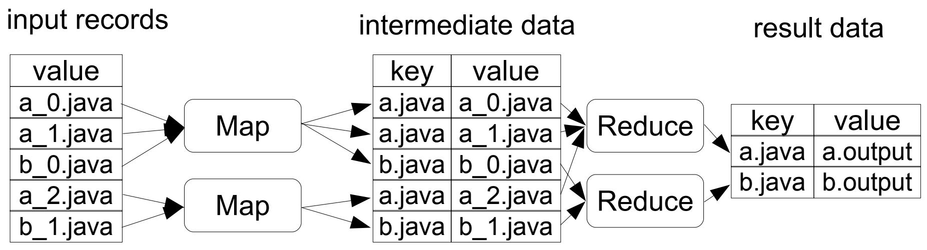

Figure 3.2: MapReduce strategy for DJ-REX.

Table 3.1: Overview of distributed steps in DJ-REX1 to DJ-REX3.

	Extraction	**Parsing**	**Analysis**
DJ-REX1	No	No	Yes
DJ-REX2	No	Yes	Yes
DJ-REX3	Yes	Yes	Yes

snapshots of $a.java$ as input, and sorts revision numbers per file: $(a.java, a\_0.java)$, $(a.java, a\_1.java)$ and $(a.java, a\_2.java)$. Pairs with the same key are then sent to the same Reducer. The final output for $a.java$ is the generated evolutionary information.

On top of this basic MapReduce strategy, we have implemented three strategies of J-REX (Table 3.1). Each strategy combines of different phases of the original J-REX implementation (Figure 3.1). The first strategy is called DJ-REX1. One machine extracts the source code offline and parses it into AST form. Afterwards, the output XML files are stored in the HDFS and Hadoop uses the XML files as input of MapReduce to analyze the change information. In this case, only 1 phase of J-REX becomes distributed. For DJ-REX2, one more phase, the parsing phase, becomes distributed. Only the extraction phase is still non-distributed, whereas the parsing and analysis phases are done inside the Reducers. Finally, DJ-REX3 is a fully distributed implementation with all the three phases in Figure 3.1 running in a distributed fashion inside each Reducer. The input for DJ-REX3 is the raw CVS data and MapReduce is used throughout all three phases of J-REX.

Table 3.2: Characteristics of Eclipse, BIRT and Datatools.

	Repository Size	**#Source Code Files**	**Length of History**	**#Revisions**
Datatools	$394MB$	$10,552$	2 years	2,398
BIRT	$810MB$	$13,002$	4 years	19,583
Eclipse	$10GB$	$56,851$	8 years	82,682

In the rest of this sub-section, we first explain our experimental environment and the details of our experiments. Then, we discuss whether or not using Hadoop for MSR studies satisfies the four requirements of Section 3.1.

3.2.3 Experimental environment

Our Hadoop installation is deployed on four computers in a local gigabit network. The four computers consist of two desktop computers, each having an Intel Quad Core Q6600 @ 2.40 GHz CPU with 2 GB RAM memory, and two server computers, one having an Intel Core i7 920 @ 2.67 GHz CPU with 4 cores and 6 GB RAM memory, and the other one having an Intel Quad Core Q6600 @ 2.40 GHz CPU with 8 GB RAM memory and a RAID5 disk. The 4 core server machine has Solid State Disks (SSD) instead of regular RAID disks. The difference in disk performance between the regular disk machines and the SSD disk server computer as measured by hdparm and iozone (64 kB block size) is shown in Table 3.3. The server's I/O speed with SSD drive is around twice as fast as the machines with regular disk for both random I/O and cached I/O.

The source control repositories used in our experiments consist of the repositories of Eclipse, BIRT and Datatools. Eclipse has a large repository with a long history, BIRT has a medium repository with a medium length history, and Datatools has a

Table 3.3: Disk performance of the desktop computer and the 4 core server computer with SSD.

	Cached read speed	**Cached write speed**
Server with SSD	$8,531 MB/sec$	$1,075 MB/sec$
Desktop	$3,302 MB/sec$	$658 MB/sec$
	Random read speed	**Random write speed**
Server with SSD	$2,986 MB/sec$	$211 MB/sec$
Desktop	$1,488 MB/sec$	$107 MB/sec$

Table 3.4: Experimental results for DJ-REX in Hadoop.

Repository	**Desktop**	**Server with SSD**	**Strategy**	**2 machines**	**3 machines**	**4 machines**
Datatools	0:35:50	0:34:14	DJ-REX3	0:19:52	0:14:32	0:16:40
BIRT	2:44:09	2:05:55	DJ-REX1	2:03:51	2:05:02	2:16:03
			DJ-REX2	1:40:22	1:40:32	1:47:26
			DJ-REX3	1:08:36	0:50:33	0:45:16
			DJ-REX3(vm)	—	3:02:47	—
Eclipse	—	12:35:34	DJ-REX3	—	—	3:49:05

small repository with a short history. Using these three repositories with different size and length of history, we can better evaluate the performance of our approach across subject systems. The repository information of the three projects is shown in Table 3.2.

3.2.4 Case study discussion

Through this case study, we seek to verify whether the Hadoop solution satisfies the four requirements listed in Section 3.1. This section uses the experiment data results of Table 3.4 to discuss whether or not the various DJ-REX solutions meet the 4 requirements outlined in Section 3.1.

Table 3.5: Effort to program and deploy DJ-REX.

J-REX Logic	No Change
MapReduce strategy for DJ-REX1	400 LOC, 2 hours
MapReduce strategy for DJ-REX2	400 LOC, 2 hours
MapReduce strategy for DJ-REX3	300 LOC, 1 hours
Deployment Configuration	1 hour
Reconfiguration	1 minute

Adaptability

Approach: We implemented three MapReduce strategies of J-REX, deployed a small cluster of machines with MapReduce platform, recorded the time we spent on MapReduce strategies implementation and deployment and counted the lines of code of the three MapReduce strategies. We explain the process to migrate the basic J-REX, a non-distributed MSR study tool, to three different strategies of MapReduce (DJ-REX1, DJ-REX2, and DJ-REX3).

Table 3.5 shows the implementation and deployment effort required for DJ-REX. We first discuss the effort devoted to porting J-REX to MapReduce. Then we present the experience of configuring MapReduce to add in more computing power. The implementation effort of the three DJ-REX solutions decreases as we got more acquainted with the technology.

Easy to experiment with various strategies

As is often the case, MSR researchers do not have the expertise required for nor do they have interest in improving the performance of their mining algorithms. The need to rewrite an MSR tool from scratch to make it run on Hadoop is not an acceptable option. If the programming time for the Hadoop migration is long (maybe as long as a major re-engineering), the chances of adopting Hadoop or other web-scale platforms become very low. In addition, if one has to modify a tool in such an invasive way, considerably more time will have to be spent to test it again once it runs in a distributed fashion.

We found that MSR study tools are very easy to port to MapReduce for the following reasons:

1. **Default classes to split data.** Hadoop provides a number of default mechanisms to split input data across Mappers. For example, the "MultiFileSplit" class splits files in a directory, whereas the "DBInputSplit" class splits rows in a database table. Often, one can reuse these existing mapping strategies.

2. **Well-defined and simple APIs.** Hadoop has well-defined and simple APIs to implement a MapReduce process. One just needs to implement the corresponding interfaces to make a custom MapReduce process.

3. **Available code examples.** Several code examples are available to show users how to write MapReduce code with Hadoop [70].

After looking at the available code examples, we found that we could reuse the code for splitting the input data by files. Then, we spent a few hours to write around 400 lines of Java code for each of the three DJ-REX MapReduce strategies. Since J-REX is developed by us, the internal structure of J-REX is very well known by us, such that the programming of DJ-REX MapReduce strategies is easy. Moreover, the programming logic of J-REX itself barely changed.

Easy to deploy and add more computing power

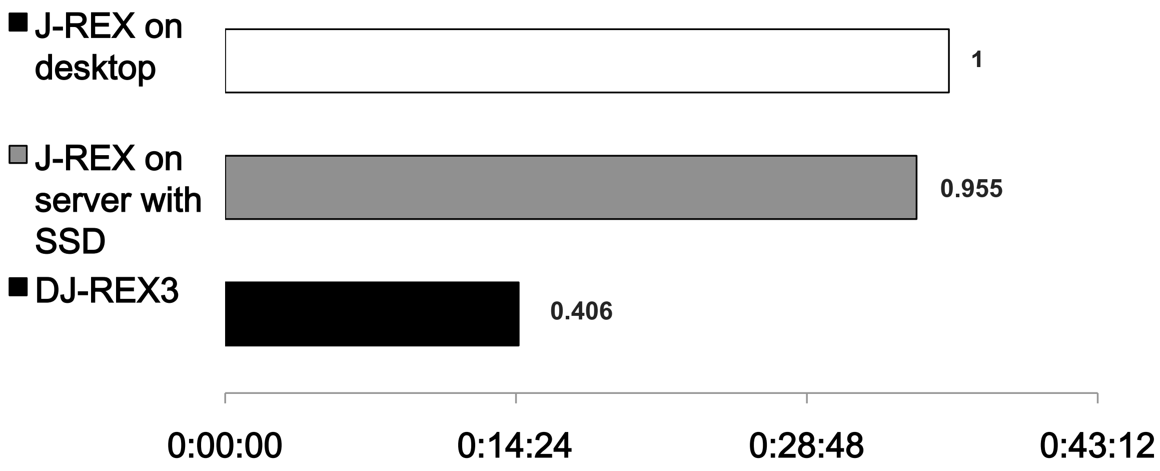

Figure 3.3: Comparing the running time of J-REX on the server machine with SSD and the desktop machine to the fastest DJ-REX3 for Datatools. In this figure, the base line is the running time of J-REX on desktop machine, which is 35 minutes and 50 seconds.

It took us only 1 hour to learn how to deploy Hadoop in the local network. To expand the experiment cluster (i.e., to add more machines), we only needed to add the machines' names in a configuration file and install MapReduce on those machines. Based on our experience, we feel that porting J-REX to MapReduce is easy and straightforward, and for sure easier and less error-prone than implementing our own distributed platform.

Efficiency

Approach: We ran J-REX with and without Hadoop on the BIRT, Datatools and Eclipse repositories. We compare the performance of all three MapReduce strategies to the performance of J-REX on desktop and server machines.

We now use our experimental data to test how much time could be saved by using MapReduce for MSR studies. Figure 3.3 (Datatools), Figure 3.4 (BIRT) and

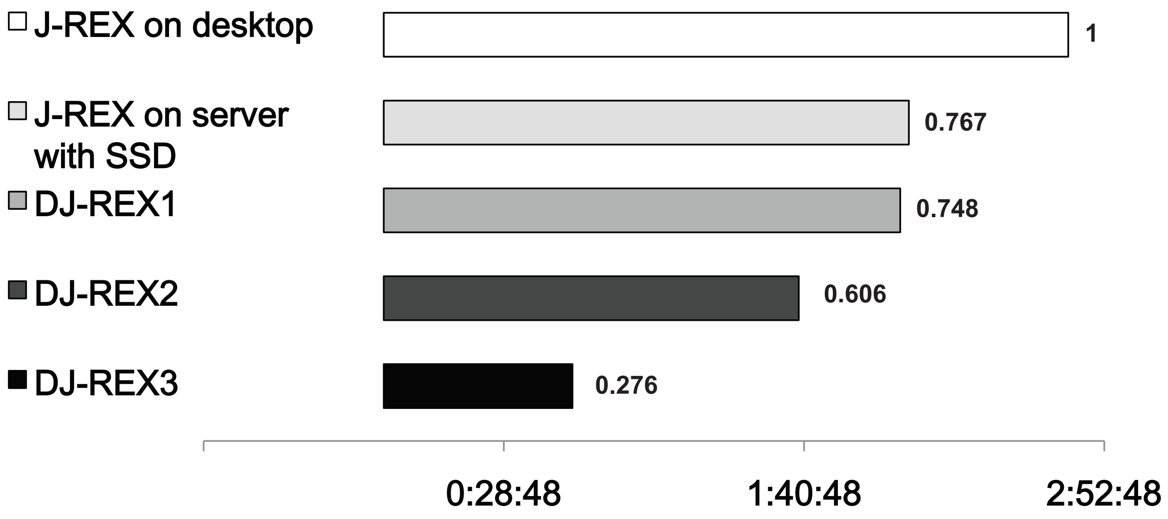

Figure 3.4: Comparing the running time of J-REX on the server machine with SSD and the desktop machine to the fastest deployment of DJ-REX1, DJ-REX2 and DJ-REX3 for BIRT. In this figure, the base line is the running time of J-REX on desktop machine, which is 2 hours, 44 minutes and 9 seconds.

Figure 3.5 (Eclipse) present the results of Table 3.4 in a graphical way. The data in the columns with title “Desktop” and “Server” in Table 3.4 are the running time of J-REX without Hadoop on our desktop machine and server machine respectively.

From Figure 3.3 (Datatools) and Figure 3.4 (BIRT), we can draw the following two conclusions. On the one hand, faster and powerful machinery *can* speed up the mining process. For example, running J-REX on a very fast server machine with SSD drives for the BIRT repository saves around 40 minutes compared with running it on the desktop machine. On the other hand, all DJ-REX solutions perform no worse but usually better than the J-REX on both desktop and server machines. As shown in Figure 3.4, the running time on the SSD server machine is almost the same to that using DJ-REX1, which only has the analysis phase distributed, since the analysis phrase is the shortest of all three J-REX phases. Therefore, the performance gain

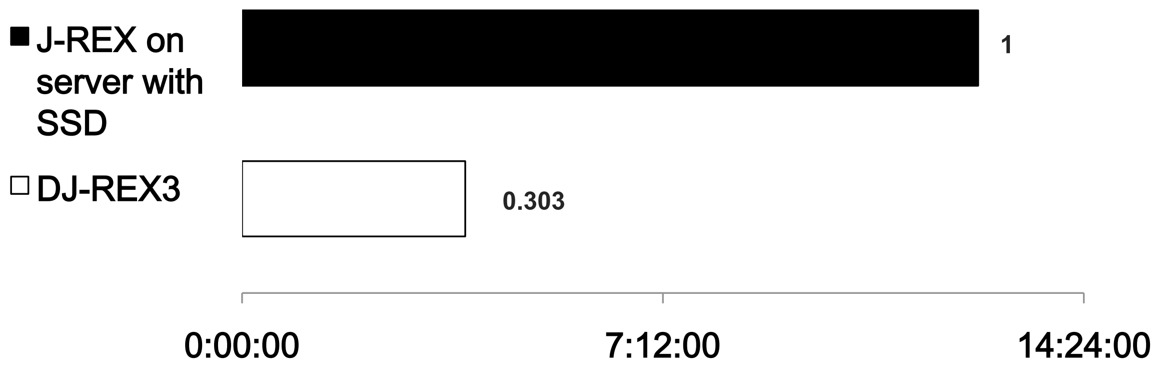

Figure 3.5: Comparing the running time of J-REX on the server machine with SSD compared to DJ-REX3 with 4 machines for Eclipse. In this figure, the base line is the running time of J-REX on a server machine with SSD, which is 12 hours, 35 minutes and 34 seconds.

of DJ-REX1 is not significant. DJ-REX2 and DJ-REX3, however, outperform the server. The running time of DJ-REX3 on BIRT is almost one quarter the time of running it on a desktop machine and one third the time of running it on a server machine. The running time of DJ-REX3 for Datatools has been reduced to around half the time taken by the desktop and server solutions, and for Eclipse (Figure 3.5) to around a quarter of the time of the server solution. It is clear that the more we distribute our processing, the less time is needed.

Figure 3.6 shows the detailed performance statistics of the three flavours of DJ-REX for the BIRT repository. The total running time can be broken down into three parts: the preprocess time (black) is the time needed for the non-distributed phases, the copy data time (light blue) is the time taken for copying the input data into the distributed file system, and the process data time (white) is the time needed by the distributed phases. In Figure 3.6, the running time of DJ-REX3 is always the shortest, whereas DJ-REX1 always takes the longest time. The reason for this is that the undistributed black parts dominate the process time for DJ-REX1 and DJ-REX2,

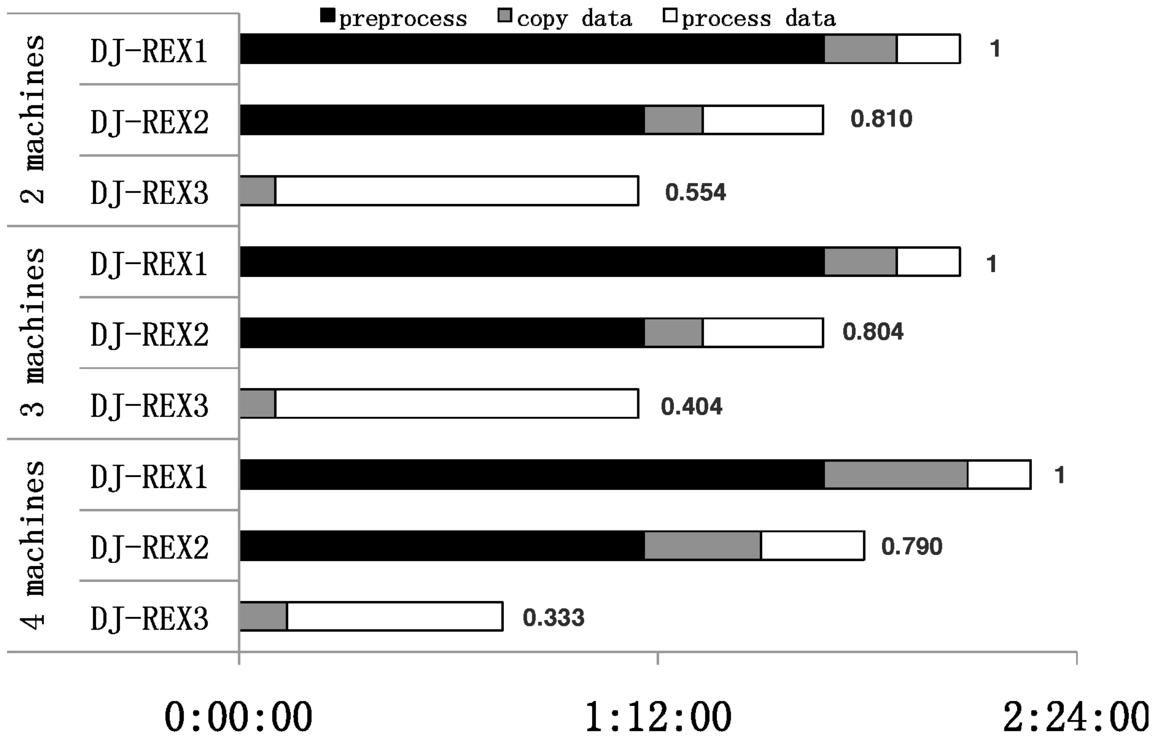

Figure 3.6: Comparison of the running time of the 3 flavours of DJ-REX for BIRT. In this figure, the base line is the running time of DJ-REX1 on 2, 3 and 4 machines. The running time of DJ-REX1 on 2, 3 and 4 machines is 2 hours, 3 minutes and 51 seconds; 2 hours, 5 minutes and 2 seconds; and 2 hours 16 minutes and 3 seconds respectively.

whereas in DJ-REX3 everything is distributed. Hence, the fully distributed DJ-REX3 is the most efficient one.

In Figure 3.6, process data time (white) is decreasing constantly. The MapReduce strategy of DJ-REX is basically dividing the job by files that are processed independently from each other in different Mappers. Hence, one could approximate the job's processing time by dividing the total processing time by the number of machines. However, the more machines there are, the smaller the incremental benefit of extra machines. A new node introduces more overhead, such as network overhead or distributed file system data synchronization. Figure 3.6 clearly shows that the time

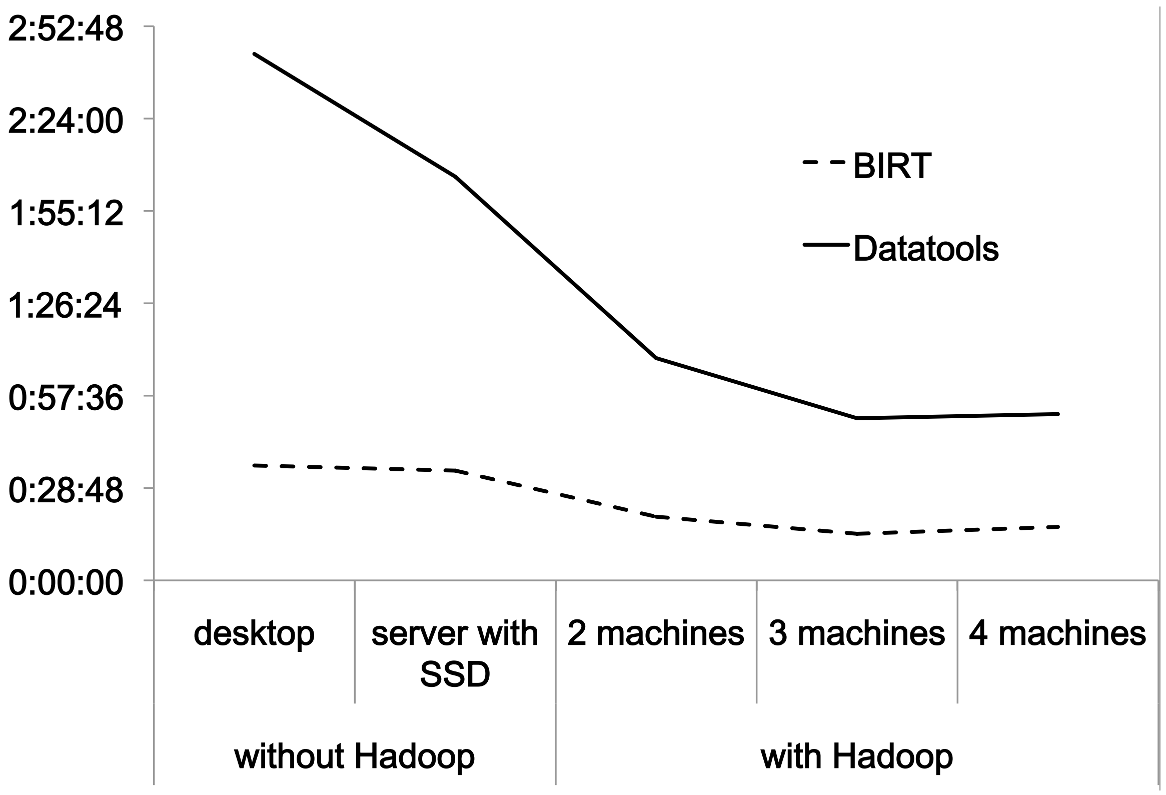

Figure 3.7: Comparison of the running time of BIRT and Datatools with DJ-REX3.

spent on copying data (grey) is increasing when adding machines and hence that the performance with 4 machines is not always the best one (e.g., for BIRT on DJ-REX2).

Our experiments show that using MapReduce to scale MSR studies is an efficient approach that can drastically reduce the required processing time.

Scalability

Approach: We ran DJ-REX3 on BIRT and Datatools with Hadoop using cluster of 2, 3 and 4 machines. We examined the scalability of the MapReduce solutions on three data repositories with varying sizes. We also examined the scalability of the MapReduce platform running on a varying number of machines.

Eclipse has a large repository, BIRT has a medium-sized repository and Datatools has a small repository. From Figure 3.3 (Datatools), Figure 3.4 (BIRT), Figure 3.5 (Eclipse) and Figure 3.6 (BIRT), it is clear that Hadoop reduces the running time for each of the three repositories. When mining the small Datatools repository, the running time is reduced to 50%. The bigger the repository, the more time can be saved by Hadoop. The running time can be reduced to 36% and 30% of the non-Hadoop version for the BIRT and Eclipse repositories, respectively.

Figure 3.7 shows that Hadoop scales well for different numbers of machines (2 to 4) for BIRT and Datatools. We did not include the running time for Eclipse because of its large data size and the fact that we could not run Eclipse on the desktop machine (we could not fit the entire data into the memory). However, from Figure 3.5 we know that the running time for Eclipse on the server machine is more than 12 hours and that it only takes a quarter of this time (around 3.5 hours) using DJ-REX3.

Unfortunately, we found that the performance of DJ-REX3 is not proportional to the used computing resources. From Figure 3.7, we observe that adding a fourth machine introduces additional overhead to our process, since copying input data to another machine out-weighs the parallelizing process to more machines. The optimal number of machines depends on the mining problem and the MapReduce strategies that are being used, as outlined in Figure 3.7.

Flexibility

Approach: We ran DJ-REX3 on BIRT with Hadoop using a cluster of 3 virtual machines. We study the flexibility of the MapReduce platform by deploying MapReduce platform on virtual machines in a multi-core environment.

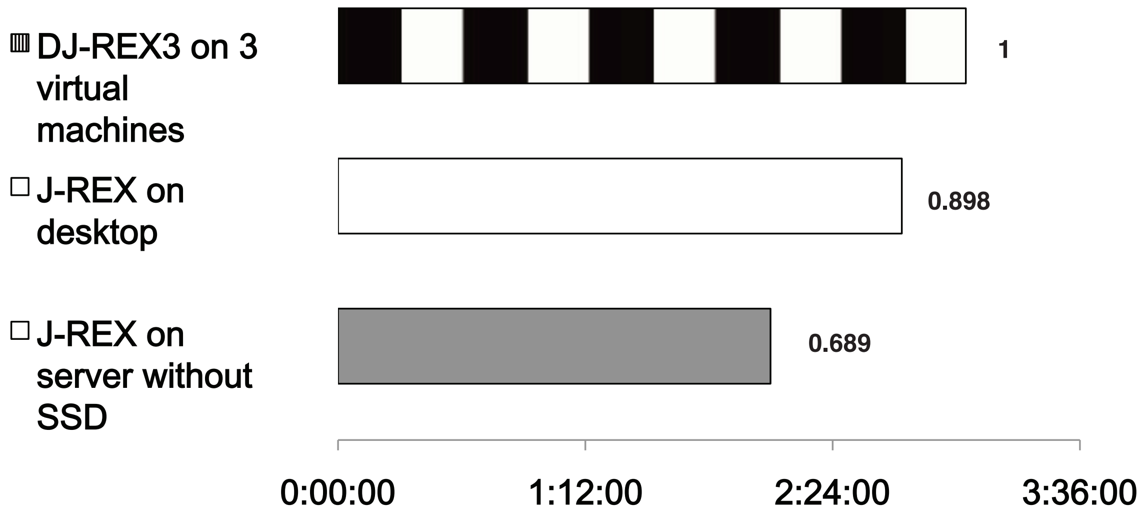

Figure 3.8: Comparing the running time of the basic J-REX on a desktop and server machine to DJ-REX-3 on 3 virtual machines on the same server machine. In this figure, the base line is the running time of DJ-REX3 on 3 virtual machines, which is 3 hours, 2 minutes and 47 seconds.

Hadoop runs on many different platforms (i.e., Windows, Mac and Unix). In our experiments, we used server machines with and without SSD drives, and relatively slow desktop machines. Because of the load balance control in Hadoop, each machine is given a fair amount of work.

Because network latency could be one of the major causes of the data copying overhead, we did an experiment with 3 Hadoop processes running in 3 virtual machines on the Intel Quad Core server machine without SSD. Running only 3 virtual machines increases the probability that each Hadoop process has its own processor core, whereas running Hadoop inside virtual machines should eliminate the majority of the I/O latency. The row with "DJ-REX3(vm)" in Table 3.4 corresponds to the experiment that has DJ-REX3 running on 3 KVM [51] virtual machines. The bar with black and white strips in Figure 3.8 shows the running time of DJ-REX3 when deployed on 3 virtual machines on the same server machine. The performance of

DJ-REX3 in virtual machines turns out to be worse than that of the undistributed J-REX. We suspect that this happens because the virtual machine setup results in slower disk accesses than deployment on a physical machine. The ability to run Hadoop in a virtual machine can be used to deploy a large Hadoop cluster in a very short time by rapidly replicating and starting up virtual machines. A well configured virtual machine could be deployed to run the mining process without any configuration, which is extremely suitable for non-experts.

3.3 Discussion and limitations

3.3.1 MapReduce on other software repositories

Multiple types of repositories are used in the MSR field, but in principle MapReduce could be used as a standard platform to speed up and scale up different analyses. The main challenge is deriving optimal mapping strategies. For example, a MapReduce strategy could split mailing list data by time or by sender name, when mining a mailing list repository. Similarly, when mapping a bug reports repository, the creator and creation time of the bug report could be used as splitting criteria. Case studies of using MapReduce on other software repositories are presented in the next chapter.

3.3.2 Incremental processing

Incremental processing is one possible way to deal with large repositories and extensive analysis. Instead of processing the data of a long history in one shot, one could incrementally process the new data on a weekly or monthly basis. However, incremental processing requires more sophisticated designs of mining algorithms, and sometimes it is just not possible to achieve. Since researchers are mostly prototyping their ideas, a brute force approach might be more desirable with optimizations (such as incremental processing) to follow later. The cost of migrating an analysis technique to MapReduce is negligible compared to the complexity of migrating a technique to support incremental processing.

3.3.3 Robustness

MapReduce and its Hadoop implementation offer a robust computation model that can deal with different kinds of failures at run-time. If certain machines fail, the Hadoop tasks belonging to the failed machines are automatically re-assigned to other machines. All other machines are notified to avoid trying to read data from the failed machines. Dean *et al.* [24] reported that MapReduce clusters with over 80 machines can become unreachable, yet the processing continues and finishes successfully. This type of robustness permits the execution of Hadoop on laptops and non-dedicated machines, such that lab computers can join and leave a Hadoop cluster rapidly and easily based on the needs of the owners. For example, students can join a Hadoop cluster while they are away from their desk and leave it on until they are back.

3.3.4 Current Limitations

Data locality is one of the most important issues for a distributed platform, as network bandwidth is a scarce resource when processing a large amount of data. To solve this problem, Hadoop attempts to replicate the data across the machines and to always locate the nearest replica of the data. In Hadoop, a typical configuration with hundreds of computers by default would have only 3 copies of the data. In this case, the chance of finding required data stored on the local machine is very small. However, increasing the number of data copies requires more space and more time to put the large amount of data into the distributed file system. This in turn leads to more processing overhead.

Deploying data into the HDFS file system is another limitation of Hadoop. In the current Hadoop version (0.19.0), all input data needs to be copied into HDFS, which introduces large overhead. As Figure 3.6 and Figure 3.7 show, the running time with 4 machines may not be the shortest one. Finding out the optimal Hadoop configuration is future work.

3.4 Chapter summary

A scalable MSR study platform should be adaptable, efficient, scalable and flexible. In this chapter, we use MapReduce to evaluate the feasibility of scaling MSR studies by the web-scale platforms. To validate our approach, we presented our experience of migrating J-REX, an evolutionary code extractor for Java, to Hadoop, an open source implementation of MapReduce. Our experiments demonstrate that our new solution (DJ-REX) satisfies the four requirements of scalable MSR study solutions. Our experiments show that running our optimized solution (DJ-REX3) on a small local area network with four machines requires 30% of time needed when running it on a desktop machine and 40% of the time on a server machine with SSD. In the next chapter, we perform more case studies to evaluate MapReduce as a general platform to scale software studies. We also document our experience in scaling several MSR experiments and evaluate the standard guidelines from the web field for MapReduce deployments.

Chapter 4

MapReduce as a General Platform for Scaling MSR Studies

In Chapter 3, we explored the use of Hadoop, a MapReduce [24] implementation, to successfully scale and speed-up J-REX, an MSR study tool. The chapter evaluates and demonstrates the feasibility of using web-scale platforms to scale MSR studies. In this chapter, we study the benefits and challenges of scaling several MSR studies using web-scale platforms. In particular, we use three representative MSR studies to demonstrate that the MapReduce platform, a scalable web analysis platform, could be used to successfully scale software studies with minimal effort. We document our experience in scaling several MSR studies, such that other researchers could benefit from our experience. Moreover, we note the changes needed to the web community's standard guidelines for the MapReduce platform when applying MapReduce to MSR studies. These changes highlight the different characteristics of MSR studies compared to web analyses and must be done to ensure that MSR researchers get the most benefit out of such a platform.

4.1 Challenges of MapReducing MSR studies.

We envision a number of important challenges based on our experience of using MapReduce in Chapter 3. We use the MapReduce example shown in Section 2.5.1 to motivate and explain these challenges. The goal of this chapter is to document our experiences addressing these challenges across various types of MSR analyses and to carefully examine the guidelines proposed by the web community regarding these challenges. By documenting the differences in analyses and data processed by both communities, we hope that our community will be able to exploit the full power of MapReduce to scale MSR studies.

4.1.1 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 requires no interaction with the other parts (i.e., a modular 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 [65]) 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 the local data (e.g., only two files), but not the whole data set. 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 [58] 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.

4.1.2 Challenge 2: Locating a suitable cluster.

Distributed platforms typically run on a large cluster of machines. Locating a suitable cluster of machines for MapReduce is a challenge. 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 [13]. 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 [10], 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.

4.1.3 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.

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. Shown in Chapter 3, 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 (in the case of commercial clusters).

4.1.4 Challenge 4: Managing data during analysis.

MapReduce needs a data management strategy to store and propagate large amounts of 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.

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

4.1.5 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.

Table 4.1: Eight types of MSR Analyses.

Name	**Description**
Metadata analysis	Analysis on metadata in software repositories, e.g., [19].
Static source code analysis	Static program analysis on source code, e.g., [31].
Source code differencing and analysis	Analysis of changes between versions of source code, e.g., [36].
Software metrics	Measuring and analyzing metrics of software repositories, e.g., [66].
Visualization	Visualizing information mined from software repositories, e.g., [59].
Clone detection methods	Detecting and analyzing similar source code fragments, e.g. [47].
Data Mining	Applying data mining techniques on software repositories, e.g., [55].
Social network analysis	Social and behavioural analysis on software repositories, e.g., [18].

4.2 Case studies

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

4.2.1 Subject systems and input data

We chose three MSR studies and associated tools to counter potential bias. Prior research identifies eight major types of MSR studies [46], as shown in Table 4.1. Techniques across these types require time-consuming processing and must cope with growing input data. We select three MSR study tools that perform six out of the eight types of MSR studies. Section 4.4 discusses the applicability of MapReduce to the two types of MSR studies 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 [2] contain the historical snapshots of every file in a software project with a log of every change during the history of the software project. As explained in Chapter 3, J-REX 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.

J-REX performs four types of MSR studies, i.e., Metadata analysis, Static source code analysis, Source code differencing and Software metrics [46].

CC-Finder. CC-Finder is a token-based clone detection tool [47] designed to extract code clones from systems developed in several programming languages (e.g., C++, and C). CC-Finder performs 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 [45]. JACK performs the Metadata analysis and Data Mining MSR studies.

Only the source code of J-REX and JACK was available to us.

4.2.2 Experimental environment

To perform our evaluation of MapReduce, we require input data, a cluster of machines and a MapReduce implementation. We repeated each experiment three times, and we report the median value of the results.

Table 4.2: Overview of the three subject tools.

	J-REX	**CC-Finder**	**JACK**
Programming Language	Java	Python	Perl
Source code available	yes	no	yes
Input data	Eclipse, Datatools	FreeBSD	Log files No. 1 & 2
Input data type	CVS repository	source code	execution log

Table 4.3: 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

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 [45]. Tables 4.2 and 4.3 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 [10]. Table 4.4 shows the configuration of the two clusters. From Chapter 3, we also have experience using a cluster in a research lab. We choose Hadoop [70] as the MapReduce implementation of our experiment.

Table 4.4: 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	$10GB$	$64GB$

4.2.3 Performance

To illustrate the scalability improvements of MapReduce for the three MSR studies in our experiments, we briefly discuss the performance obtained using MapReduce in our experiments, compared to the performance on a single machine without MapReduce. A more detailed analysis of the performance gains of MapReduce for J-REX can be found in Chapter 3. Table 4.5 shows the best performance measurement 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, the running time reported by [47] for detecting code clones in the FreeBSD source code on one machine is 40 days. We did not repeat that experiment. From the table, we can see on a cluster of 10 machines (SHARCNET), that the running time of J-REX and JACK is reduced by a factor 9 and 6 respectively. For CC-Finder, the running time is 59 hours. Our experiments show that MapReduce is able to perform large-scale MSR studies.

4.3 Migration experiences

While the previous section confirms that MapReduce can effectively scale several types of MSR studies, it took us several attempts and experiments to achieve such

Table 4.5: Best results for the migrated MSR tools.

Tool name	**Input data**	**One machine**	**MapReduce version**	**Relative running time ratio**	**Cluster**
J-REX	**Eclipse**	755min	80min	0.106	SHARCNET
CC-Finder	**FreeBSD**	–	59$hours$	–	student lab
JACK	**Log file No.1**	580min	98min	0.169	SHARCNET

Table 4.6: Challenges of MapReducing MSR problems.

Challenge number	**Challenge description**
1	Migrating MSR tools to a divide-and-conquer programming model.
2	Locating a suitable cluster.
3	Optimizing MapReduce strategy design and cluster configuration.
4	Managing data during analysis.
5	Recovering from errors.

performance results. In this section we distill our experience such that others can benefit from them. For each challenge shown in Table 4.6, we discuss our findings and provide advice based on our experience. We also compare our findings relative to common guidelines provided by the web community.

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

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

J-REX. For J-REX, we broke down the one-shot processing of a whole repository such that every single file in the repository 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 every 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: *"a.java/a_0.java"*, *"a.java/a_1.java"* and *"a.java/a_2.java"*. The Reducer generates *"a.java/evolutionary information of a.java"*. The implementation details are shown in the previous chapter as DJ-REX3.

CC-Finder. Our MapReduce implementation adopts the same computation model as D-CCFinder [53], which consists of the following steps:

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. Mapper sends the pair to a Reducer.
4. The Reducer invokes CC-Finder on a particular pair to run the clone analysis.

Figure 4.1 shows an example of the computational model for detecting code clones in 4 files. From the figure, we can see that every file needs to be compared to every other file and to itself, resulting into 10 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. As we can see, this MapReduce approach only breaks down the input and runs analysis on them separately, without any aggregation.

Similar approaches of only using "Map" or "Reduce" are found in examples of MapReduce strategies such as "Distributed Grep" [24]. Other analyses, such as the

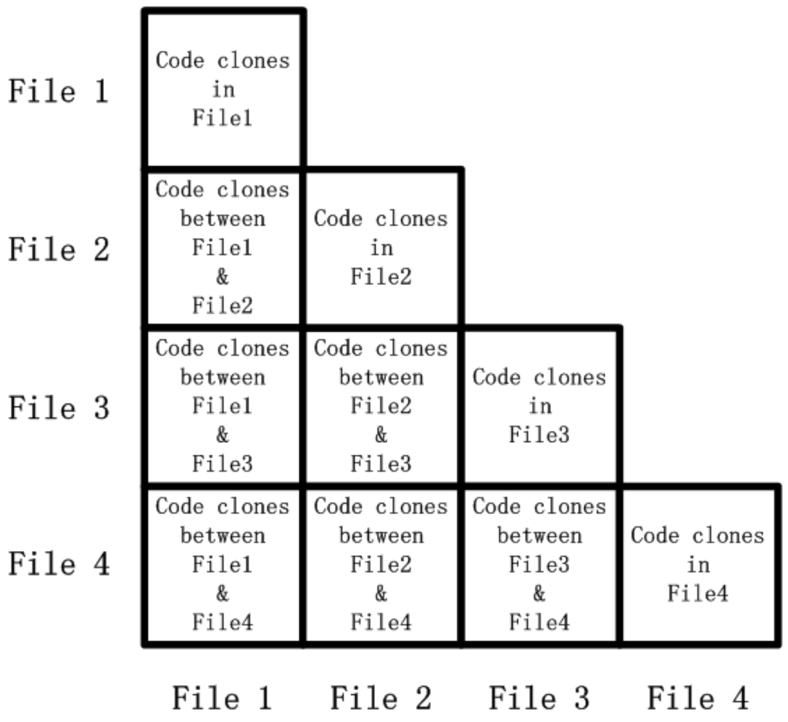
File 1
File 2
File 3
File 4
Code clones in File1
Code clones between File1 & File2
Code clones in File2
Code clones between File1 & File3
Code clones between File2 & File3
Code clones in File3
Code clones between File1 & File4
Code clones between File2 & File4
Code clones between File3 & File4
Code clones in File4
File 1
File 2
File 3
File 4

Figure 4.1: Example of the typical computational model of clone detection techniques.

example LOC study in Section 4.1, perform analysis in both “Map” and “Reduce” phases.

Notable Findings.

We summarize below our main observations.

1. **Locality of analysis.** The majority of MapReduce uses in the web community 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 only one single file is required for one analysis. 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. Yet, all tools show good performance after

being MapReduced. For CC-Finder we adopted the computation model proposed by [53] using the services provided by the MapReduce platform instead of spending considerable time implementing the needed 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 our "Mappers" mapped all these files 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 analysis. 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 the previous chapter, we used a four-machine MapReduce cluster in our research lab. In this chapter, 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 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 community. 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 heterogeneous infrastructures to provide a homogeneous cluster. The virtual machine solution works well for non-I/O intensive analysis and as *playground* for MSR studies 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: Optimizing MapReduce strategy design and cluster configuration.

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 community) 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 amount. The fine-grained breakdown had each part processing 200MB while the coarse-grained breakdown had each part processing 1GB of data (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 power needed for each fine-grained unit has a large portion of the time wasted on communication overhead.

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 4.2, the top value corresponds to the maximum running time across all the machines, which determines the running time of the whole MapReduce process. The higher the grey box in the violin plot, the less balanced the workload of machines (higher variance in workload).

We then increased the number of Hadoop tasks to 100 for J-REX and 54 (the number of files, shown in Table 4.3) 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 4.2 show that the running time of every machine after increasing the number of Hadoop tasks is more balanced than before (higher 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 than with fewer Hadoop tasks.

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 [2]. CVS repositories store the history of each file in a separate file, leading to a large number of input files.

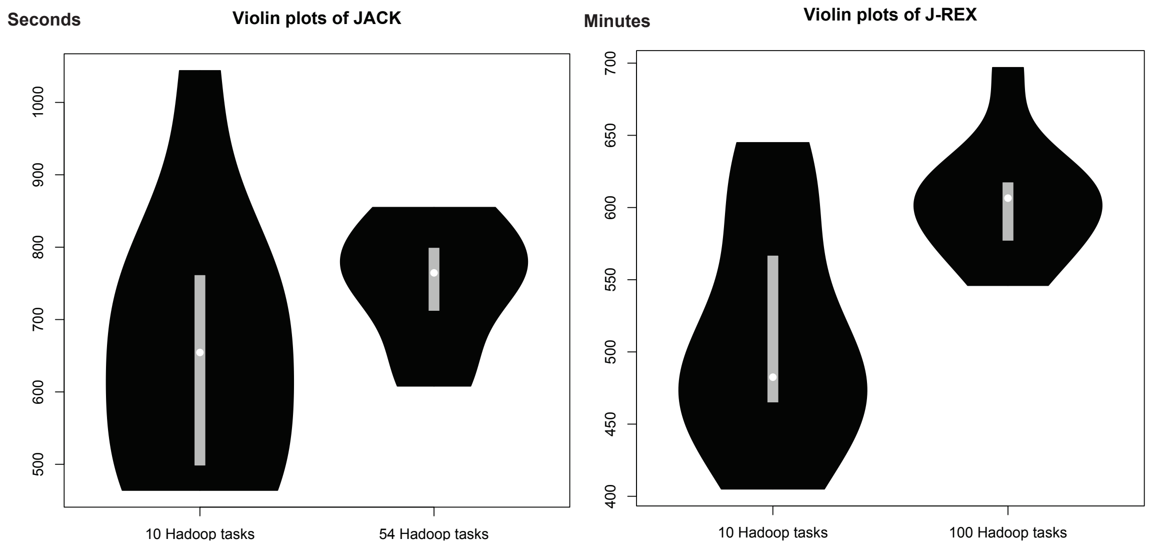

Figure 4.2: Violin plots of machine running-time for JACK and J-REX.

As shown in Table 4.2, 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 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. Figure 4.3 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 bad performance of JACK. In Figure 4.3, JACK does not improve its performance when moving from 6 machines to 10 machines for analyzing Log file No. 1. 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.

Notable Findings.

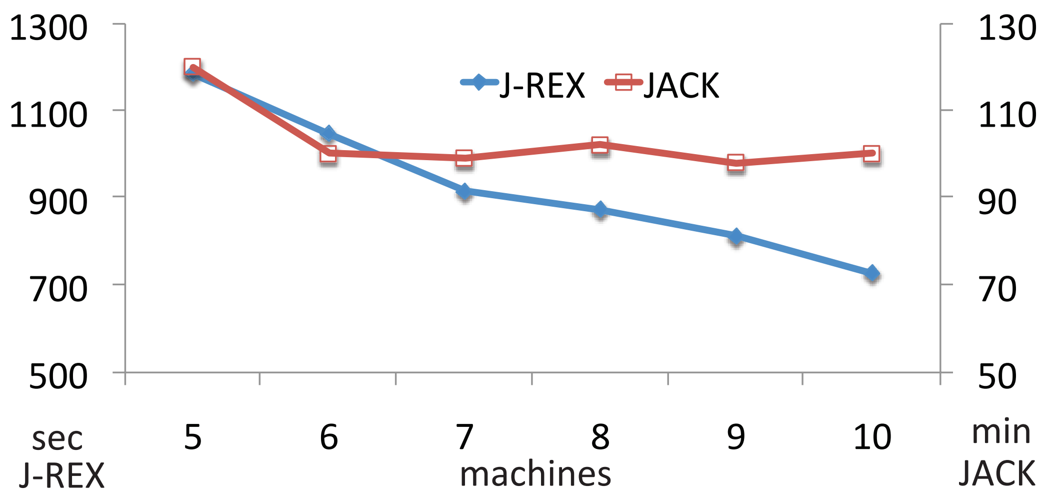

Figure 4.3: Running time trends of J-REX and JACK with 5 to 10 machines.

The web community 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 relatively limited processing and that they vary in locality. On the one hand, we would suggest that researchers analyze files in groups instead of individually in order to reduce platform overhead. However, the grouping of files might cause an 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 MSR studies on balanced input data benefit more from more machines in the cluster than small-scale MSR studies 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 the middle from 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 Map or Reduce processes that can run simultaneously on one machine, which is typically the number of cores of the machines [6].

This recommendation works well for web analysis, which is traditionally fine-grained. Fine-grained MSR study tools like J-REX, which have a large number of input key/value pairs, can still adopt these recommendations. Coarse-grained MSR study 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, e.g., the number of input files for JACK. Because the "Map" procedures are identity functions in our case studies, the number of "Map" procedures should be small to avoid the overhead.

Challenge 4: Managing data during analysis.

Our experiments 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 unfortunately optimized for reading and perform poorly for writing data [70]. 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 then must retrieve after the MapReduce run is completed. However, we have to take the risk of losing output data and having to re-perform 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 efficient data writes. As MSR studies may generate large amounts of result data, saving the result data in HDFS may introduce much overhead. From our experience, we recommend: 1) the use of the local file system if the result data of an MSR tool consists of a large amount of data; and 2) the use of HDFS if the result data is small in size.

Challenge 5: Recovering from errors.

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 "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 study 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., [45]) for detecting possible failures of the execution of an MSR study tool.

Table 4.7: 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, [53]
Data Mining	Challenge 1	hard	yes, [7]
Social network analysis	Challenge 1	medium	yes, [12]

4.4 Applicability

This section discusses the applicability of MapReduce to other MSR studies than the ones we considered in our case studies. We examine the eight types of MSR studies presented in Section 4.2. The descriptions and examples of the eight types of MSR studies are presented in Table 4.1. 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 4.7 the main challenges of migration, the ease of migration and the existence of prior research of scaling the analysis.

Metadata analysis. In metadata analysis, data can be broken down by the type of the metadata. For example, source code repository can be broken down to every source code file and bug repository can be broken down to bug report.

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) collect the required data in the other source code files; 2) perform 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. The MapReduce strategies of J-REX are presented in Section 3.2.

Software metrics. The MapReduce strategies can be designed based on the types of software metric. For example, software complexity metrics for different revisions can be generated by splitting the historical data by changes. Our example of studying the evolution of #LOC of a software project in Section 2.5.1 is another example.

Visualization. The visualization techniques that we consider consist of a regular MSR technique, followed by the generation of a visualization. For example, Adams *et al.* [14] study the evolution of the build system of Linux kernel and generate visualizations of the build dependency graph. The generation of visualizations is often non-modular, so it is hard to migrate them to MapReduce.

Clone-detection methods. Clone-detection techniques are non-modular. This is the reason why they are hard to migrate to MapReduce. [53] proposes an approach to map clone-detection to divide-and-conquer, which we adopted in our case study as MapReduce strategy. The strategy is presented in Section 4.2.

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 the Data Mining algorithms on Hadoop [7].

Social network analysis. Social networks can be analyzed as a graph with nodes and edges. Some of the analysis of the entire graph can be broken down to analyses of every node or edges. X-RIME [12] is a Hadoop library for social network analysis.

Based on the examination of the eight types of MSR studies, most studies are able to migrate to MapReduce, despite some challenges. Moreover, previous research (e.g., [7, 12, 53]) has addressed some of the more challenging migration problems.

4.5 Limitations and threats to validity

We present the threats to validity for the findings in this chapter.

4.5.1 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. We firmly believe that other MSR tools would benefit from adoption of web-scale platforms. Our case studies provide promising findings and we encourage other researchers to explore MapReducing their tools. Section 4.4 provides a brief discussion of generalization across other MSR studies.

4.5.2 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 report the median value of the results.

4.5.3 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 thesis, 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.

4.6 Chapter summary

MSR studies continue to analyze 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 chapter, we demonstrate that standard web scale platforms, like MapReduce, could generally be used to effectively and efficiently scale MSR studies, despite several challenges. We document our experiences such that others can 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 community guidelines. We hope that our experiences will help others explore the use of web scale platforms to scale software studies, instead of having to develop their own solutions. However, designing and implementing “Map” and “Reduce” strategies requires additional effort and are not easy to re-use. In the next chapter, we explore using Pig in practice to improve the re-usability of scaling MSR studies by web-scale platforms.

Chapter 5

Large-Scale MSR Studies with Pig

In Chapter 3 and 4 we showed that we could scale and speed-up MSR studies using MapReduce. However, our prior experience highlighted some of the limitations of MapReduce as a platform for large-scale MSR studies. In particular, the use of the MapReduce platform required in-depth knowledge of the MapReduce processing phases and required additional effort of designing Map and Reduce strategies for every data processing phase (DJ-REX1, DJ-REX2, DJ-REX3). The implemented MapReduce strategies are hard to re-use in other MSR studies. In this chapter, we explore the use of Pig, a popular web-scale platform, for performing large-scale MSR studies. Through three case studies we carefully demonstrate the use of Pig to prepare (i.e., ETL) software data for further analysis. Our experience shows that Pig programs have a modular design for combining and re-using ETL modules in MSR studies.

5.1 Motivating example of data preparation in a large-scale MSR Study

In this section, we present an example of data preparation in a typical MSR study, taken from our real life experience. The ETL tool we use in this example is J-REX, i.e., the highly optimized MSR tool for software evolution study introduced in Section 3.2.

Researcher Lily wanted to study software defects and code clone on a large, long-lived software project.

After we used J-REX to prepare the data Lily required, she found out that she needed the list of methods and their source code content for every snapshot of the extracted data. In addition, she requested information that lists, for every snapshot, those methods that were added or deleted. We changed J-REX to provide the newly required information.

In order to study software defects in the data that we prepared for her, Lily wanted to use a heuristic that relates changes in source code to bugs by checking keywords in the commit logs of the source control system. Since the commit log data was not extracted yet, we performed the data extraction for the commit log data.

While performing data analysis on the prepared data, Lily found that the source code content of some methods was missing. She also required the deletion of methods to be associated with the first snapshot after the deletion instead of with the snapshot of deletion. For example, if method *foo* was in snapshot *1.0* but not in snapshot *1.1*, Lily needed the deletion of *foo* to be recorded for snapshot *1.1*, not for snapshot *1.0*.

Because of the large scale of the data, we spent much effort on fixing the bug, prepared the new data, and delivered it to Lily. She made great progress in her research, but now she had to perform clone detection on the extracted method content source code. As we are not clone detection experts, we chose to use an existing clone detection tool. Even though we already extracted the source code of methods, we now had to output the source code as intermediate data in the format used by the clone detection tool. After doing the clone detection, we needed to collect the result and indicate if a method in the method list contains code clones.

To conclude, the data preparation process takes a large amount of time because the many iterations of ETL on large-scale data. In order to gather all required data, we had to rely on the expertise of existing tools. Additional effort is required for fixing bugs in the data preparation platforms.

5.2 Requirements for ETL in large-scale MSR studies

From the basic requirements presented in Section 3.1 for a general web-platform that scales MSR studies, our further experience of performing large-scale MSR studies using MapReduce and the motivating example presented in Section 5.1, we identify three requirements for data preparation in large-scale MSR studies.

5.2.1 Modular design

Data preparation for large-scale MSR studies requires modular design because of the following two reasons:

1. **Module re-uses.** Various basic approaches and algorithms are widely used in

MSR studies, such as extracting CVS logs and grouping source code changes into transactions. Modular design of data preparation platforms enables reusing such software modules and building them into a new tool chain instead of re-developing the same functionality in different MSR studies.

2. **Expertise re-uses.** Preparing data for MSR studies requires the combination of information from various types of software repositories with different methodologies. However, researchers are not experts in every field of MSR. Hence, the data preparation platform requires modular design to enable combining existing tools. In Section 5.1, we combine a source control data extractor, bug report data extractor and also the output data of a clone detection tool. Effort is spent on combining different tools and merging the output data of different tools.

In our experience, most platforms for data preparation enable modular design. Kenyon [20] is a data extractor for different source control systems. Kim *et al.* combined the extracted data by Kenyon with CC-Finder [47], a code clone detector, and a location tracker that tracks code clones across versions [49] to perform Clone Genealogy Analysis [49].

5.2.2 Scalability

Scalability is required for the data ETL in MSR studies because of two reasons:

1. **Data size and growth.** Software repositories contain massive amounts of data. Hence, studying software engineering data is very time consuming. For example, performing clone detection on the FreeBSD source code may take more than a month [53]. Some large-scale data preparation cannot be performed

because of hardware limitations. However, data from software engineering activities keeps on growing in size. Scalability is necessary to study the massive amounts of growing data.

2. **Iterative analysis.** The pipeline illustrated in Figure 2.1 shows the iterative process for studying software engineering data. The iterative analysis requires the data preparation to be scalable, such that each iteration finishes in a reasonable amount of time.

A number of approaches are available to address the requirement of scalability. Ad hoc distributed programs are developed to scale data preparation in general-purpose languages. D-CCfinder [53] is an example that scales CC-Finder [47] to run in an ad hoc distributed environment to support large-scale clone detection. MapReduce based data preparation platforms are scalable, because as a distributed framework, MapReduce benefits from the scalability of a distributed environment. Parallel databases provide scalability for SQL-like data preparing platforms. However, most Prolog-like platforms are hard to scale. Erlang [21] is a Prolog-like language that is able to run on distributed environment to scale [16], but the distributed Erlang relies on Message Passing techniques [32] and requires much programming effort.

5.2.3 Debuggability

As ETL needs to deal with huge data sets having irregular data formats and often missing data, bugs can easily slip into extraction, transformation and loading scripts. To debug ETL scripts, researchers need to be able to examine the data generated during the data preparation process. Two important requirements for data debugging are:

1. **Intermediate data.** Every step of data preparation generates intermediate data. Storing intermediate data is important for data debugging because being able to inspect the correctness of intermediate data facilitates locating problems in the data preparation process and avoids having to start the data preparation all over again.

2. **Data sampling and previewing.** Being able to preview the results of data preparation on a sample of the data before processing all of the data is required to avoid having to re-run costly data preparation tools because of a bug in the last set of processed data. For example, heuristics on source control system logs can be used to group source code changes into different categorizations [41], such as feature introducing changes and bug fixing changes. Researchers would like to automatically have a sample of data for each categorization to see if their heuristics work well before starting to perform the heuristic on the whole data.

Traditional platforms can all support debuggable data preparation. However, as MapReduce relies on data storage in a distributed environment, intermediate data of MapReduce is saved as a list of string pairs distributed in the distributed file system. Examining intermediate data of MapReduce, and sampling and previewing output data of MapReduce requires reading the data via distributed file system API, or copying the data into local file system.

As shown in Table 5.1, traditional techniques can meet all requirements with relative ease, except for scalability, while MapReduce excels at scalability, but falls short at the other two requirements. In the next section, we introduce Pig [60], a data processing platform and a high level programming language on top of MapReduce to combine the advantages of the different platforms for large-scale MSR studies.

Table 5.1: How do traditional software study platforms meet the three requirements for data preparation?

	Modular	**Scalable**	**Debuggable**
SQL-like	-	o	o
Prolog-like	+	-	+
Regular programming	+	-	+
MapReduce	o	+	o
Legend	+ requirement is met o requirement is met with additional effort - requirement is not met or is hard to meet		

5.3 Case studies

This section uses Pig to perform data preparation (ETL) on three software studies. For each study, we show what data is required for the analysis and how we implemented the data's preparation with Pig. After we present the three MSR studies and our implementations, we discuss our experience of performing data preparation for large-scale MSR studies with Pig.

5.3.1 Data preparation for three MSR studies

We first present the data prepared for each MSR study and our implementations using Pig to perform the data preparation. The subject system for our three MSR studies is *Eclipse*, a widely used Java IDE as shown in Table 3.2.

Study one:

The first MSR study is an empirical study on the correlation between updating comments in the source code and the appearance of bugs.

Required data: This analysis requires the following data for every change in the source control system:

1. is the change related to a bug?
2. does the change update comments?

Implementation: The first step of implementing a Pig program is to break down the ETL into a number of program units. The following program units are used:

1. Loading data from a CVS repository into Pig storage as a (*file name*, *file content*) pair.
2. Generating log data for every source code file.
3. Generating version information for every source code file.
4. Using heuristics to check if a change is related to bugs.
5. Extracting every version of source code for every source code file.
6. Transforming every snapshot of every source code file into XML format.
7. Checking comment changes of every version of every source code file.

The Pig Latin source code for study one is shown in Figure 5.1. The corresponding Java code for every Pig program unit is shown in Appendix A.1, A.2, A.3, A.13, A.4, A.5 and A.7.

In the Pig Latin scripts of study one shown in Figure 5.1, line 1 loads the content of every file from the input data. Line 2 generates the CVS log data for every file and line 3 generates the historical versions from the CVS log data. Line 5 and line 6 check if a change is related to bugs. The variable "BUGCHANGES" generated in

```
1  CVSMETADATA = load 'EclipseCvsData'
       using ExtPigStorage() as (
       filename:chararray, filecontent:
       chararray);
2  HISTORYLOG = foreach CVSMETADATA
       generate ExtractLog(filename,
       filecontent);
3  HISTORYVERSIONS = foreach HISTORYLOG
       generate ExtractVersions($0);
4
5  BUGCHANGES= filter HISTORYVERSIONS by
        IsBug{$0};
6  NOBUGCHANGES=filter HISTORYVERSIONS
       by not IsBug{$0};
7
8  CODE = foreach HISTORYVERSIONS
       generate ExtractSourceCode($0);
9  XMLS = foreach CODE generate
       ConvertSourceToXML($0);
10 COMMENTEVO= foreach XMLS generate
       EvoAnalysisComment($0);
11 BUGRESULT= join BUGCHANGES by $0.$0,
       COMMENTEVO by $0.$0;
12 NOBUGRESULT= join NOBUGCHANGES by $0.
       $0, COMMENTEVO by $0.$0;
13
14 dump BUGRESULT;
15 dump NOBUGRESULT;
```

Figure 5.1: Pig Latin script for study one.

line 5 consists of the changes related to bugs and the variable "NOBUGCHANGES" generated in line 6 consists of the changes not related to bugs. Line 8 extracts every snapshot of all the source code files from the input data by the historical versions generated in line 3. These snapshots of source code files are transformed to XML files by line 9. Line 10 analyzes the evolution of comments of of every source code file. Line 11 and line 12 join the evolution of comments, i.e., output of line 10, with the changes related and not related to bugs respectively.

Study two:

The second MSR study is an empirical study on software defects in both cloned and non-cloned methods in a software system, which is actually the motivating example presented in Section 5.1.

Required data: This analysis requires the following data for every file at every revision:

1. is the revision related to a bug?
2. (for every method in the file) is the method new or has it been deleted?
3. source code for every method.
4. (for every method) is the method cloned?

Implementation:

Because of the modular programming style of Pig, we re-used program units 1, 2, 3, 4, 5, 6 from study one without any modification. In addition, we also need program units for:

1. Checking which methods have been added or deleted in every version of every source code file.
2. Generating every method's content.
3. Clone detection on all method content.
4. Ruling out falsely reported cloned methods.

Moreover, the intermediate data, e.g., variable *CODE* in line 8 of the Pig Latin source code of study one shown in Figure 5.1, can be stored in HDFS and re-used in the other studies. The Pig Latin script for study two, which re-uses the existing

```
1  METHODEVO= foreach XMLS generate EvoAnalysisMethod($0);
2  METHODCONTENTS= foreach CODE generate GetMethod($0);
3
4  METHODPAIRS= cross METHODCONTENTS, METHODCONTENTS;
5
6  CLONES= foreach METHODPAIRS generate CloneDetection($0);
7
8  CLONES= filter CLONES by TimeOverlap($0);
9
10 BUGRESULT= join BUGCHANGES by $0.$0, CLONES by $0.$0, METHODEVO by $0.$0;
11 NOBUGRESULT= join NOBUGCHANGES by $0.$0, CLONES by $0.$0, METHODEVO by $0.$0;
12
13 dump BUGRESULT;
14 dump NOBUGRESULT;
```

Figure 5.2: Pig Latin script for study two.

variables from study one is shown in Figure 5.2. The corresponding Java code for every Pig program unit is shown in Appendix A.8, A.9, A.10 and A.11.

In the Pig Latin scripts of study two shown in Figure 5.2, line 1 analyzes the evolution of methods in every source code file. Line 2 generates the source code content of every method in all the source code files. To perform clone detection, line 4 generates cross product of method content and itself. The cross products consist of pairs of method content, such that line 6 can perform clone detection on each pair of method content. Running clone detection on all source code files that ever existed may falsely report code clones between parts of the source code that never existed at the same point in time. Line 8 filters out those false code clones. Line 10 and line 11 join the evolution data of methods, the result of code clone detection, and historical versions related and not related to bugs respectively.

Study three:

In the third study, we prepare data to calculate the complexity of the changes in periods. Hassan uses this data to predict software defects [38].

Required data: This study requires the number of changed LOC in Feature Introduction Modification (FI) changes, i.e., changes that introduce new features for every time span.

Implementation:

Study three re-uses program units 1, 2 and 3 in study one. Three more program units are required:

1. Checking for every change if it is an FI change.
2. Grouping changes in every time span. In particular, we use quarters in 2008 as time spans.
3. Counting changed #LOC.

The corresponding Pig Latin script, which uses the variables from study one and two is shown in Figure 5.3. And the corresponding Java code for every Pig program unit is shown in Appendix A.14, A.12 and A.6.

In the Pig Latin scripts of study three shown in Figure 5.3, line 2 uses five specific days to indicate the four quarters in 2008 as time spans and line 4 uses the key value generated by line 2 as *"$8"* to group the commits into time spans. Line 5 counts the changed #LOC of every group of changes generated by line 4.

```
FIVERSIONS= filter HISTORYVERSIONS by
     IsFI($0);
TIMESPANS = foreach FIVERSIONS
    generate TimeSpan($0, (
    "2008/01/01" , "2008/04/01" ,
    "2008/07/01" , "2008/10/01" ,
    "2009/01/01" ));

TIMESPAN_GROUP = group TIMESPANS by
    $8;
CHANGEDLOC_TIMESPAN= foreach
    TIMESPAN_GROUP generate ChangeLOC
    ($0);
dump CHANGEDLOC_TIMESPAN;
```

Figure 5.3: Pig Latin script for study three.

5.3.2 Experience Report

We now discuss our experiences with Pig to perform data preparation in the three three MSR studies.

Modular Design

Pig stimulates a modular design in which each Pig program is decomposed into a number of small program units. Pig Latin composes the whole program by combining the program units together. With such modular programming style, adding a new program unit or changing one program unit does not affect other program units. Program units in Pig are re-usable for data preparation of different MSR studies.

The program units we identified for the three MSR studies are summarized in Table 5.2. The source code details of these program units are shown in Appendix A. Many program units are re-used in all of the three case studies. Figure 5.4 shows how we composed different program units into the data preparation process of the three MSR studies. The numbers in the program unit boxes in Figure 5.4 correspond to the

Table 5.2: Program units for case studies.

Appendix number	**Program unit name**	**Description**
Appendix A.1	**ExtPigStorage**	Loading data into Pig.
Appendix A.2	**ExtractLog**	Generating CVS repository log.
Appendix A.3	**ExtractVersions**	Parsing CVS log to generate historical versions.
Appendix A.4	**ExtractSourceCode**	Extracting source code files.
Appendix A.5	**ConvertSourceToXML**	Converting source code to XML format.
Appendix A.6	**ChangeLOC**	Counting number of changed LOC for a source code file.
Appendix A.7	**EvoAnalysisComment**	Comment evolution analysis.
Appendix A.8	**EvoAnalysisMethod**	Method evolution analysis.
Appendix A.9	**GetMethod**	Generating method content.
Appendix A.10	**CloneDetection**	Detecting clones on program entity pairs.
Appendix A.11	**TimeOverlap**	Ruling out false clones.
Appendix A.12	**TimeSpan**	Checking if a change is in a time period.
Appendix A.13	**IsBug**	Checking if a change is related to a bug.
Appendix A.14	**IsFI**	Checking if a change is a feature introducing (FI) change.

appendix number in Appendix A. The most widely re-used program units provide basic functionalities of MSR studies.

Scalability

In Chapter 3, we verified the feasibility of using MapReduce to prepare data for MSR studies. Our experiments show that using Hadoop (an open-source MapReduce implementation) on a four-machine cluster improves the computation time by 30-40% when analyzing the CVS [2] source control repository of the Eclipse project.

Based on the two reasons of the requirement of scalability presented in Section 5.2, Pig should decrease both the "raw" running time of MSR studies and the time spent on iterative analysis. However, we want to examine the relative scalability of Pig compared to MapReduce.

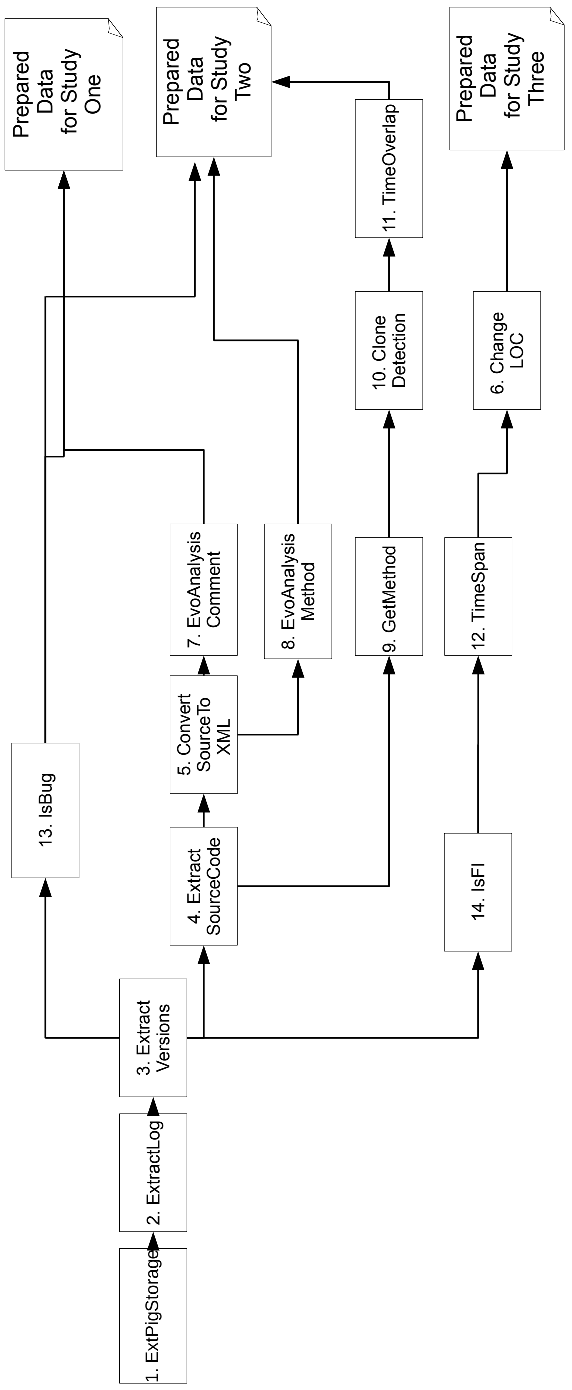

Figure 5.4: Composition of the data preparation process for the three MSR studies performed with PIG.

Table 5.3: Configuration of the server machine and the distributed computing environment.

	Server machine	**distributed computing environment**
# Machines	1	5
CPU	16 × Intel(R) Xeon X5560 (2.80GHZ)	8 × Intel(R) Xeon E5540 (2.53GHz)
Memory	64GB	12GB
Network	Gigabit	Gigabit
OS	Ubuntu 9.10	CentOS 9.10
Disk type	SSD	SATA

Table 5.4: Running time of J-REX on single machine, Hadoop platform and Pig platform. The base line of the relative running time ratio is the running time of J-REX on single machine.

Sub-folder name	**On single machine**	**On Hadoop platform**	**On Pig platform**
runtime	12 min	2 min (0.167)	1.5 min (0.125)
e4	164 min	20 min (0.122)	23 min (0.140)
pde	240 min	16 min (0.067)	24 min (0.100)

To evaluate the scalability of Pig, we perform an MSR study on three pieces of input data with the non-distributed MSR tool, the MapReduce platform and the Pig platform. We first used J-REX to prepare data from three major sub-folders of *Eclipse* CVS repositories on a powerful server machine (see Table 5.4). We then ran J-REX on both the Hadoop platform and the Pig platform. The Hadoop and Pig platforms are deployed in our private distributed computing environment. The configuration of the distributed computing environment is shown in Table 5.3. The performance of the MSR study is shown in Table 5.4. Using the running time of the original J-REX as base line, the numbers in the brackets show the relative running time ratio of both Hadoop J-REX and Pig J-REX.

The running time in Table 5.4 of the Pig-based J-REX turned out to be almost the same as the running time of the MapReduce-based J-REX. Even though the original J-REX ran on a very powerful server, the Pig-based J-REX is much faster than the original J-REX. These findings seems to confirm recent research findings that showed that the running time of a Pig program is around 1.3 times as long as the running time of native Hadoop [26]. However, in our experiments, we found that Pig sometimes could be more efficient than the original J-REX and the MapReduce-based J-REX. This can be explained by the fact that additional I/O is introduced when the original J-REX and the MapReduce-based J-REX store intermediate data, while the Pig-based J-REX does not have to store intermediate data during the data preparation.

As shown in the data pipeline in Section 2.2 and the motivating example in Section 5.1, MSR studies require iterative analysis. Without Pig, MSR researchers may need to run the whole experiment for MSR studies again and again for every iteration or they have to design the formats and methods to load and store intermediate data. With Pig, all intermediate data is stored as variables, which MSR researchers can store and load without designing and implementing data formats and additional methods. Based on our experience, the iterative analysis is well supported by Pig and the time spent on iterative MSR is much shorter than performing MSR studies with original MSR tools.

From the above experiences, we consider the Pig platform to provide efficient scalability for data preparation of large-scale MSR studies.

Debuggability

In our case studies, Pig proved to be a debuggable platform.

When the result of data analysis seemed incorrect, we checked the intermediate data generated by every program unit in the data preparation process. Pig stores this intermediate data as variables in the cloud environment. The intermediate data can easily be checked and used without additional effort.

Since MSR studies are performed on large-scale data and researchers mostly only need a small sample of data to examine, data sampling and previewing is important. Similar to SQL, Pig Latin has keywords *LIMIT* and *SAMPLE* to select a limited sample of the data. *Pigpen* is a data previewing tool that is released together with Pig [60]. *Pigpen* samples data in representative way and shows output of Pig program on those sample data. We used *Pigpen* to check the correctness of Pig Latin script. Once the script seemed to work on the *Pigpen* data, we did not have to run any more test runs on it.

5.4 Discussion

In this section, we discuss about the other possible web scale data processing platforms that can prepare data for large-scale MSR studies, and the limitation of Pig in performing MSR studies.

5.4.1 Pig/MapReduce or Parallel DBMS

MapReduce is a distributed framework and is also a simple programming model that processes massive amounts of data [24]. Pig is a high-level programming language on top of MapReduce. Parallel DBMS are the database systems that are able to run on computer clusters [61]. Storing data and performing query on a cluster of machines, Parallel DBMS is much more scalable than normal DBMS that can only be deployed on one machine. Two different opinions about Pig/MapReduce exist. One opinion considers Pig/MapReduce to be a simple programming platform that allows programmers without parallel and distributed programming experience to build distributed programs. A large number of programs have been built using Pig/MapReduce in Google [25] and Hadoop [70], an open-source MapReduce implementation, has been used widely in various fields.

However, as an opposite point of view, Dewitt *et al.* [8] posted a blog article to criticize that Pig/MapReduce is a "major step backward", because it does not provide several essential advantages of databases. Comparative analysis of MapReduce to Parallel DBMS has been performed [61, 67]. The comparison shows the trade-offs between MapReduce and Parallel DBMS. One interesting finding is that the initial data loading process of Parallel DBMS took much longer than loading data into the distributed file system of MapReduce, while the observed analysis time of Parallel DBMS was shorter than MapReduce. Researchers suggest that DBMS should be used on data that is not changed or re-loaded often [61]. The iterative analysis MSR studies may require data loading multiple times. The slow data loading process of Parallel DBMS may become a bottle neck of the scalability of the MSR studies.

Dean *et al.*, the authors of the first paper about MapReduce [24], claimed in [25] that the comparison between MapReduce and Parallel DBMS in [61] is based on flawed assumptions. One of the interesting issues Dean *et al.* pointed out is that MapReduce is a flexible data processing tool that does not require loading data into a database. The data analyzed by MapReduce does not require a schema. The requirement of data schema explains why Parallel DBMS is faster in data analysis but much slower in loading data.

5.4.2 Pig or MapReduce

In Chapter 3 and 4, we verified the feasibility of using MapReduce to support MSR research as a general framework. As a high-level language on top of MapReduce, Pig has several unique benefits.

1. **Increasing productivity:** The productivity of using Pig is much higher than programming with the native MapReduce paradigm. Programmers of Yahoo! claimed that 10 lines of Pig Latin code provided the functionality of 200 lines of Java code. In our case studies, we used less than 20 lines of Pig script for every MSR study and on average less than 100 lines of Java code for every program unit, such that over 1,000 lines of code are developed for the three studies in Section 5.3. In Chapter 3, only 400 lines of Java code are required to migrate J-REX to MapReduce. To our knowledge, we are the first to report the additional development effort involved to implement user-defined Pig program units. However, this effort is not as high in practice, as the Java code of a program unit contains a large part of boiler-plating code for Java class and method declarations. In addition, reusing program units reduces the program effort and increases the productivity of Pig.

2. **Easy migration:** Migrating an existing MSR study tool to MapReduce requires fitting the existing algorithms and processes into the MapReduce programming paradigm. Researchers may not want to change their way of solving problems to the paradigm of MapReduce. However, instead of migrating existing tools to MapReduce, Pig co-ordinates tools as modules in a sequential way that is more natural for programmers. Since researchers do not want to spend effort in migrating, Pig is better than MapReduce for migrating.

According to our experiences shown in Section 5.3, Pig improves the maintainability and reusability of MSR tools with minimal performance overhead. We consider Pig a better choice to perform large-scale MSR studies.

5.4.3 Pig or Hive

Hive is a data warehouse on top of Hadoop [5]. As one of the most similar and alternative techniques to Pig, Hive also uses the MapReduce infrastructure provided by Hadoop. Hive uses an SQL-like language, while Pig uses a sequential language. Database connection APIs, for example JDBC and ODBC, can connect to Hive and perform analyses as clients.

Similar to using SQL on Parallel DBMS, the advantage of Hive is to put structure into the data and query with a query language. However, to perform data preparation (ETL) with unstructured and flexible software engineering data, Pig seems to be a better solution than Hive.

5.4.4 Data loading and retrieving

Although Pig satisfies the three requirements for data preparation, it still has its limitations. As Pig runs on top of Hadoop, the input data of a Pig program needs to be loaded into HDFS and the data prepared by Pig needs to be copied from HDFS to the local file system for further analyses. Because software engineering data is typically large, loading and retrieving data is an important limitation of Pig for data preparation of large-scale MSR studies with Pig.

However, this limitation is not unique to Pig. The main alternative to Pig, Parallel DBMS, also has to perform data loading. Research shows that loading data into

MapReduce data storage, i.e. HDFS, is much faster than loading data into parallel database [61, 67]. As data generated by Pig can be stored in HDFS, it can be viewed by the HDFS programming interfaces provided by Hadoop. Moreover, data retrieving is not necessary if further analyses on the data are programmed in Pig or MapReduce. Such that this limitation of Pig does not compromise the using of Pig to enable large-scale MSR studies.

5.5 Chapter summary

Traditional software analysis platforms are used to perform large-scale MSR studies with ever larger and more complex data. Even though MapReduce is capable to scale MSR studies as a general platform, the migrating process requires additional design and programming effort, and is hard to re-use in practice. In this chapter, we adopt Pig, a high-level data processing programming language on top of MapReduce, to improve the re-usability in scaling MSR studies using web-scale platforms. We used Pig to prepare (i.e., Extract, Transform, and Load) software data for three MSR studies. From our experience, Pig meets the requirements of data preparation for studying software data because of its modular design, the scalability provided by Pig's underneath infrastructure (MapReduce) and its debuggability. We believe that our experiences and our implementation of Pig program units for preparing data for three MSR studies will be valuable for other researchers and practitioners.

Chapter 6

Conclusion

This chapter summarizes the main ideas presented in this thesis. In addition, we propose some future work to ease the use of web-scale platforms for large-scale MSR studies.

MSR research requires scaling to ever larger and more sophisticated analyses. Existing solutions for scaling MSR research are mostly ad hoc and hard to maintain. Given that need for scalable studies is very prominent in the MSR field. We believe the MSR field can benefit from web-scale platforms to overcome the limitations of current approaches. To evaluate our hypothesis, we perform large-scale MSR studies using web-scale platforms such as MapReduce and Pig in this thesis. Through our case studies, we conclude that MapReduce can effectively and efficiently scale MSR studies as a general platform, despite several challenges. Moreover, Pig can be used to improve the re-usability in practice. We documented the experiences and lessons we learnt from scaling MSR study tools. These experiences, lessons learnt and our source code are valuable for scaling MSR studies for other researchers and practitioners.

6.1 Major topics addressed

Chapter 3 performs a study of the feasibility of using MapReduce, a web-scale platform, to scale an MSR tool called J-REX. From our case study, we find that the running time of J-REX on a four-machine cluster after migrated to Hadoop is 30% to 40% of the original J-REX's running time on one machine. This chapter illustrates that MapReduce platform, as an example of web-scale platforms, is feasible to enable a large scale MSR study of software evolution.

Chapter 4 uses MapReduce as a general platform to scale MSR studies. From the experience of Chapter 3, we generalize five challenges of scaling MSR studies by MapReduce. We document our experience of addressing the challenges. Moreover, we evaluate the standard guideline from the web field in our case studies and find that the guidelines from the web field need to be changed in MSR studies to avoid sub-optimal performance. This chapter shows that MapReduce, as an example of web-scale platforms, can be used as a general platform to enable different types of MSR studies.

Finally, Chapter 5 presents using Pig, a web-scale platform on top of Hadoop, to improve the re-usability and maintainability of migrating MSR studies to web-scale platforms. Through the case studies of perform data ETL in three MSR studies, we show that Pig can make the scaling of MSR studies more re-usable and easier to maintain. We document our code in the thesis and the appendix to assist other MSR researchers and practitioners. This chapter shows that Pig platform can be used to improve the re-usability and maintainability of using web-scale platforms to enable large-scale MSR studies.

6.2 Thesis contributions

The contributions of this thesis are as follows:

1. We verified the feasibility and benefits of scaling MSR experiments using the MapReduce and Pig platforms. Our experiments show that running the MapReduce version of J-REX on a small local area network with four machines requires 30% to 40% of the running time of the original J-REX. On a cluster of 10 machines (SHARCNET), the running time of MapReduce version of J-REX and JACK is reduced by a factor 9 and 6 respectively. For CC-Finder, the running time is also decreased significantly.

2. We documented our experiences in scaling MSR studies and provided code samples of program units in Pig, such that other researchers could benefit from our experience and code samples. The code samples are documented in Appendix A. Our experiences suggest that:

   (a) Migrating local and semi-local analyses is much simpler than migrating non-local analysis.

   (b) We have explored the use of virtual machines on heterogeneous infrastructures to provide a homogeneous cluster. The virtual machine solution works well as *playground* for analysis and debugging before deployment on larger clusters.

   (c) Large-scale MSR studies on balanced input data benefit more from more machines in the cluster than small-scale MSR studies with unbalanced input data.

   (d) We find that Hadoop's error recovery mechanism enabled us to have *agile clusters* with machines joining and leaving the cluster based on need.

   (e) Pig platform can improve the re-usability and maintainability of the migration of MSR studies to web-scale platforms.

3. We also note that changes are needed to the Web community's standard guidelines for the MapReduce platform when migrating MSR analyses to web-scale platforms. These changes highlight the different characteristics of MSR analyses and must be done to ensure that MSR researchers get the most benefits from such web-scale platforms. In particular, the different characteristics noted are as following.

   (a) A majority of MapReduce uses in the web community are local in nature, while for our case study we find that MSR studies may be non-local or semi-local, which requires more efforts to migrate to MapReduce.

   (b) Heterogeneous infrastructures are not frequently used in the web community. In the software engineering research community, heterogeneous infrastructures are the norm rather than the exception.

   (c) The standard guidelines for MapReduce platform works well for web analysis, which is traditionally fine-grained. Fine-grained MSR study tools like J-REX, which have a large number of input key/value pairs, can still adopt these recommendations. Coarse-grained MSR study tools like JACK, which have a small number of input key/value pairs, should not adopt these recommendations.

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

6.3 Future research

Our first future plan is to perform more case studies of different types of software engineering studies on web-scale platforms. Based on our case studies, we plan to build a Java library of MapReduce strategies and Pig program units, which provides various MSR algorithms and techniques readily to use on top of Hadoop and Pig.

Second, we plan to find approaches that can assist in determining the most optimal configurations for different MSR studies in different hardware environments.

Third, most of the “Map” phases of our MapReduce strategies only pass the data to “Reduce” phases, but do not process the data. In our future work, we plan to find out if we can use both “Map” and “Reduce” phases to improve the performance of MSR studies or if MSR tools do not benefit from both “Map” and “Reduce” phases conceptually.

In addition, since CC-Finder [47] was not open-source software when we performed our case studies in Chapter 4 and became open-source recently, we plan to carefully examine the source code of CC-Finder and develop CC-Finder using MapReduce or Pig from scratch.

Finally, there are other web-scale platforms that can possibly support large-scale MSR studies. We are interested in exploring the benefits and shortcomings of other web-scale platforms. We also plan to use SQL-like web-scale platforms, such as Hive [5], to perform SQL-base MSR analysis after the data ETL is performed by Hadoop.

Bibliography

[1] Amazon EC2. https://aws.amazon.com/ec2/.

[2] CVS. http://www.cvshome.org/.

[3] Eclipse JDT. http://www.eclipse.org/jdt.

[4] Hadoop distributed file system. http://hadoop.apache.org/hdfs/.

[5] Hive. http://hadoop.apache.org/hive/.

[6] How Many Maps And Reduces. http://wiki.apache.org/hadoop/HowManyMapsAndReduces.

[7] MAHOUT. http://lucene.apache.org/mahout/.

[8] MapReduce: A major step backwards. http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/.

[9] Self-service, prorated super computing fun! http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun/.

[10] SHARCNET. https://www.sharcnet.ca.

[11] Vertica home page. http://www.vertica.com.

[12] X-RIME home page. http://xrime.sourceforge.net/.

[13] Anurag Acharya, Guy Edjlali, and Joel Saltz. The utility of exploiting idle workstations for parallel computation. *SIGMETRICS Perform. Eval. Rev.*, 25(1):225–234, 1997.

[14] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. The evolution of the linux build system. *Electronic Communications of the ECE-ASST*, 8, February 2008.

[15] Giulio Antoniol, Massimiliano Di Penta, and Mark Harman. Search-based techniques applied to optimization of project planning for a massive maintenance project. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 240–249, Washington, DC, USA, 2005. IEEE Computer Society.

[16] J. Armstrong, R. Virding, M. Williams, and C. Wikstroem. *Concurrent programming in ERLANG*. Citeseer, 1993.

[17] Sushil Bajracharya, Joel Ossher, and Cristina 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.

[18] Olga Baysal and Andrew 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.

[19] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. Extracting structural information from bug reports. In *MSR '08: Proceedings of the Fifth International Working Conference on Mining Software Repositories*, May 2008.

[20] Jennifer Bevan, E. James Whitehead, Jr., Sunghun Kim, and Michael Godfrey. Facilitating software evolution research with kenyon. In *ESEC/FSE '05: Proceedings of the 10th European Software Engineering Conference*, 2005.

[21] F. Cesarini and S. Thompson. *Erlang programming.* O'Reilly Media, 2009.

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

[23] R. Chandra. *Parallel programming in OpenMP.* Morgan Kaufmann, 2001.

[24] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51, 2008.

[25] Jeffrey Dean and Sanjay Ghemawat. MapReduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.

[26] Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a high-level dataflow system on top of Map-Reduce: the Pig experience. *Proc. VLDB Endow.*, 2(2):1414–1425, 2009.

[27] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP '03: Proceedings of the 19th ACM symposium on Operating Systems Principles*, 2003.

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

[29] Michael W. Godfrey and Qiang Tu. Evolution in open source software: A case study. In *ICSM '00: Proceedings of the International Conference on Software Maintenance*, page 131, Washington, DC, USA, 2000. IEEE Computer Society.

[30] Jesus M. Gonzalez-Barahona, Gregorio Robles, Martin Michlmayr, Juan José Amor, and Daniel M. German. Macro-level software evolution: a case study of a large software compilation. *Empirical Softw. Engg.*, 14(3):262–285, 2009.

[31] Carsten Görg and Peter 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.

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

[33] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1):10–18, 2009.

[34] Mark 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.

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

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

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

[38] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 78–88, Washington, DC, USA, 2009. IEEE Computer Society.

[39] Ahmed E. Hassan and Ric C. Holt. Using development history sticky notes to understand software architecture. In *IWPC '04: Proceedings of the 12th IEEE International Workshop on Program Comprehension*, 2004.

[40] Ahmed E. Hassan and Richard C. Holt. Studying the evolution of software systems using evolutionary code extractors. In *IWPSE '04: Proceedings of the Principles of Software Evolution, 7th International Workshop*, pages 76–81, Washington, DC, USA, 2004. IEEE Computer Society.

[41] Ahmed E. Hassan and Richard C. Holt. Using development history sticky notes to understand software architecture. In *IWPC '04: Proceedings of the 12th IEEE International Workshop on Program Comprehension*, page 183, Washington, DC, USA, 2004. IEEE Computer Society.

[42] Ross Ihaka and Robert Gentleman. R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

[43] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, 2007.

[44] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. An automated approach for abstracting execution logs to execution events. *J. Softw. Maint. Evol.*, 20(4):249–267, 2008.

[45] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder 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.

[46] Huzefa Kagdi, Michael L. Collard, and Jonathan 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.

[47] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro 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.

[48] C.J. Kapser and M.W. Godfrey. Cloning considered harmful considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, 2008.

[49] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 187–196, New York, NY, USA, 2005. ACM.

[50] Colin Kirsopp, Martin J. Shepperd, and John 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.

[51] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux virtual machine monitor. In *OLS '07: The 2007 Ottwa Linux Symposium*, pages 225–230, 2007.

[52] Meir M. Lehman and Juan F. Ramil. Software evolution–background, theory, practice. *Information Processing Letters*, 88(1-2):33 – 44, 2003. To honour Professor W.M. Turski's Contribution to Computing Science on the Occasion of his 65th Birthday.

[53] Simone Livieri, Yoshiki Higo, Makoto Matushita, and Katsuro 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.

[54] Angela Lozano and Michel Wermelinger. Assessing the effect of clones on changeability. In *ICSM '08: Proceedings of 24th IEEE International Conference on Software Maintenance*, pages 227–236, Beijing, China, October 2008. IEEE.

[55] Walid Maalej and Hans-Jorg 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.

[56] K. Maruyama and S. Yamamoto. A tool platform using an XML representation of source code information. *IEICE Transactions on Information and System E Series D*, 89(7):2214, 2006.

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

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

[59] Allen P. Nikora and John 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.

[60] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew 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.

[61] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 165–178, New York, NY, USA, 2009. ACM.

[62] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Sci. Program.*, 13(4):277–298, 2005.

[63] Gregorio Robles and Jesus M. Gonzalez-Barahona. Developer identification methods for integrated data from various sources. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, 2005.

[64] Gregorio Robles, Jesus M. Gonzalez-Barahona, Martin Michlmayr, and Juan Jose Amor. Mining large software compilations over time: another perspective of software evolution. In *MSR '06: Proceedings of the 3rd International workshop on Mining software repositories*, 2006.

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

[66] Yonghee Shin, Robert Bell, Thomas Ostrand, and Elaine 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.

[67] Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. MapReduce and parallel DBMSs: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.

[68] E. Van Emden and L. Moonen. Java quality assurance by detecting code smells. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering*, page 97, Washington, DC, USA, 2002. IEEE Computer Society.

[69] Panos Vassiliadis, Alkis Simitsis, and Spiros Skiadopoulos. Conceptual modeling for ETL processes. In *DOLAP '02: Proceedings of the 5th ACM international workshop on Data Warehousing and OLAP*, pages 14–21, New York, NY, USA, 2002. ACM.

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

Appendix A

Sample Source Code of Pig Programming Units

A.1 ExtPigStorage

```
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import org.apache.pig.builtin.PigStorage;
import org.apache.pig.data.DataByteArray;
import org.apache.pig.data.Tuple;
import org.apache.pig.impl.io.BufferedPositionedInputStream;

/**
 * @author Ian Shang
 * @version 1.0
 * Pig Storage Class to read content of every file..
 */
public class ExtPigStorage extends PigStorage {

        private String currentFile;
        private long offset;
        private long end;
        ByteArrayOutputStream mBuf = null;
        /**
         * @author Ian Shang
         * @version 1.0
         * @param filename: the name of the input file.
         * @param in: input stream
         * @param offset: offset
         * @param end: end
         * @return null
         * bind input file to input stream
         */
        public void bindTo(String fileName,
            BufferedPositionedInputStream in,
                        long offset, long end) throws IOException {
                currentFile = fileName;
                this.offset = offset;
                this.end = end;
                super.bindTo(fileName, in, offset, end);
        }
        /**
         * @author Ian Shang
         * @version 1.0
         * @return Tuple of next data record
         */
        public Tuple getNext() throws IOException {
                if (in == null || in.getPosition() > end) {
                        return null;
                }
                //only get java,v file
                if (!currentFile.endsWith("java,v"))
                        return null;
                List<Object> newList = new ArrayList<Object>();
                newList.add(currentFile);

                byte[] array = new byte[(int) (end - offset)];
                in.read(array);
                in = null;
                ArrayList<Object> mProtoTuple = new ArrayList();
                mProtoTuple.add(new DataByteArray(array));

                newList.add(mProtoTuple.toString());
                Tuple tuple = mTupleFactory.newTupleNoCopy(newList);
                return tuple;
        }
}
```

A.2 ExtractLog

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.DefaultTupleFactory;
import org.apache.pig.data.Tuple;
import org.apache.pig.impl.util.WrappedIOException;

/**
 * @author Ian Shang
 * @version 1.0
 * evaluation UDF to use rlog to extract change log of CVS
     repositories.
 */
public class ExtractLog extends EvalFunc<Tuple> {
        /**
         * @author Ian Shang
         * @version 1.0
         * @param Tuple with a ,v file
         * @return Tuple with CVS log of ,v file.
         */
        public Tuple exec(Tuple input) throws IOException {
                if (input == null || input.size() == 0)
                        return null;
                try {
                        String name = (String) input.get(0);
                        String content = (String) input.get(1);
                        if (content.startsWith("["))
                                content = content.substring(1);
                        if (content.endsWith("]"))
                                content = content.substring(0,
                                    content.length() - 1);

                        // store the content to local
                        //hack with hdfs data naming schema
                        String home = System.getProperty("user.home"
                            );
                        if (name.startsWith("file:"))
                                name = name.substring(6);
                        if (name.startsWith("hdfs"))
                                name = name.substring(name.indexOf("
                                    9000") + 4);
                        if (name.startsWith("/"))
                                name = name.substring(1);
                        name = home + name;
                        File file = new File(name);
                        File parent = new File(file.getParent());
                        parent.mkdirs();
                        FileWriter fw = new FileWriter(file);
                        fw.write(content);
                        fw.close();
                        //use J-REX library to extract CVS log
                        String rlog = JrexUtil.JrexUtil.extractRlog(
                            name);
                        Tuple tname = DefaultTupleFactory.
                            getInstance().newTuple();
                        tname.append(name);
                        tname.append(rlog);
                        return tname;
                } catch (Exception e) {
                        throw WrappedIOException.wrap(
                                        "Caught exception processing
                                            input row ", e);
                }
        }
}
```

A.3 ExtractVersions

```
import java.io.IOException;
import java.util.List;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.DataBag;
import org.apache.pig.data.DefaultBagFactory;
import org.apache.pig.data.DefaultTupleFactory;
import org.apache.pig.data.Tuple;
import org.apache.pig.impl.util.WrappedIOException;

/**
 * @author Ian Shang
 * @version 1.0
 * evaluation UDF to extract commits from the CVS log.
 */
public class ExtractVersions extends EvalFunc<DataBag> {
        /**
         * @author Ian Shang
         * @version 1.0
```

```
     * @param Tuple with CVS log of a Java source code file
     * @return Tuple with commits of a Java source code file
     */
    public DataBag exec(Tuple input) throws IOException {
        if (input == null || input.size() == 0)
            return null;
        try {
            Tuple inputtuple = (Tuple) input.get(0);
            String name = (String) inputtuple.get(0);
            String content = (String) inputtuple.get(1);
            //use J-REX library to extract commits from
               the CVS log
            List<String[]> rlog = JrexUtil.JrexUtil
                            .ExtractCommitsbyRlog(
                               content);
            DataBag commits = DefaultBagFactory.
               getInstance().newDefaultBag();
            for (String[] commit : rlog) {
                Tuple committuple =
                   DefaultTupleFactory.getInstance()
                              .newTuple();
                for (String s : commit) {
                    committuple.append(s);
                }
                commits.add(committuple);
            }
            Tuple tname = DefaultTupleFactory.
               getInstance().newTuple(name);

```

```
                        DataBag output = DefaultBagFactory.
                            getInstance().newDefaultBag();
                        output.add(tname);
                        output.addAll(commits);
                        return output;
                } catch (Exception e) {
                        throw WrappedIOException.wrap(
                                        "Caught exception processing
                                           input row ", e);
                }

        }
}
```

A.4 ExtractSourceCode

```
import java.io.IOException;
import java.util.Iterator;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.DataBag;
import org.apache.pig.data.DefaultBagFactory;
import org.apache.pig.data.DefaultTupleFactory;
import org.apache.pig.data.Tuple;

/**
 * @author Ian Shang
 * @version 1.0
 * evaluation UDF to extract source code from CVS repositories.
 */
public class ExtractSourceCode extends EvalFunc <DataBag>
{
        /**
         * @author Ian Shang
         * @version 1.0
         * @param Tuple with a databag of every commits of a Java
            file.
         * @return Tuple with extracted source code snapshots of a
            Java file.
         */
    public DataBag exec(Tuple input) throws IOException {
                Iterator<Tuple> databag = ((DataBag) input.get(0)).
                    iterator();

                Tuple tname = (Tuple) databag.next();
                String filename = (String) tname.get(0);
                // caculate cvsroot
                String[] tmp = filename.split("/");
                String cvsroot = new String();
                for (int i = 0; i < tmp.length; ++i) {
                        cvsroot = cvsroot + tmp[i] + "/";
                        if (tmp[i].equals("cvs")) {
                                cvsroot = cvsroot + tmp[(i + 1)];
                                break;
                        }
                }

                DataBag output = DefaultBagFactory.getInstance().
                    newDefaultBag();
                output.add(tname);

                while (databag.hasNext()) {// for every snapshot
                        Tuple commit = databag.next();
                        String version = (String) commit.get(0);
                        //use J-REX library to extract source code
                        String content = JrexUtil.JrexUtil.
                            callCvsGetCode(version,
                                        filename, cvsroot);
                        if (content != null && content != "") {
                                Tuple sourcetuple =
                                    DefaultTupleFactory.getInstance()
                                            .newTuple();
                                sourcetuple.append(version);
                                sourcetuple.append(content);
                                output.add(sourcetuple);
                        }
                }
                return output;
    }
}
```

A.5 CovertSourceToXML

```
import java.io.IOException;
import java.io.StringWriter;
import java.util.Iterator;
import java.util.List;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.DataBag;
import org.apache.pig.data.DefaultBagFactory;
import org.apache.pig.data.DefaultTupleFactory;
import org.apache.pig.data.Tuple;
import org.w3c.dom.Document;
import JrexUtil.JavaFile;

/**
 * @author Ian Shang
 * @version 1.0
 * evaluation UDF to convert Java source code to XML representative.
 */
public class ConvertSourceToXML extends EvalFunc <DataBag>
{
        /**
         * @author Ian Shang
         * @version 1.0
         * @param Tuple with a databag of Java source code.
         * @return Tuple with converted XML data of java source code
            .
         */
```

```
    public DataBag exec(Tuple input) throws IOException {
        Iterator<Tuple> databag = ((DataBag) input.get(0)).
            iterator();
        Tuple sub = null;
        List<Object> sublist = null;
        if (input.size() > 1) {
            sub = (Tuple) input.get(1);
            sublist = sub.getAll();
        }

        Tuple tname = (Tuple) databag.next();
        String filename = (String) tname.get(0);

        DataBag output = DefaultBagFactory.getInstance().
            newDefaultBag();
        output.add(tname);

        Tuple xmltuples = DefaultTupleFactory.getInstance().
            newTuple();

        while (databag.hasNext()) {
            Tuple commit = databag.next();
            //get source code of every commit of the
                Java file
            String version = (String) commit.get(0);
            String sourcecode = (String) commit.get(1);
            //use J-REX library to parse Java source
                code to a Java Class
            JavaFile jf = JrexUtil.ParseJava.
                parseJavaCode(sourcecode,
                            filename, sublist);
            if (jf == null) {
                    return null;
            }

            jf.setDocument();
            Document document = jf.getDocument();
            //transform the Java source code information
                 to XML format
            TransformerFactory tf = TransformerFactory.
                newInstance();
            Transformer transformer;
            try {
                    transformer = tf.newTransformer();

                    DOMSource source = new DOMSource(
                        document);
                    transformer.setOutputProperty(
                        OutputKeys.ENCODING, "UTF-8");
                    transformer.setOutputProperty(
                        OutputKeys.INDENT, "yes");

                    StringWriter stringw = new
                        StringWriter();
                    StreamResult result = new
                        StreamResult(stringw);
```

```
                                        transformer.transform(source, result
                                            );
                                        document = null;
                                        String returnS = stringw.toString();
                                        //use the XML data to create tuple
                                        Tuple xmltuple = DefaultTupleFactory
                                            .getInstance().newTuple();
                                        xmltuple.append(version);
                                        xmltuple.append(returnS);
                                        xmltuples.append(xmltuple);
                              } catch (Exception e) {
                                        e.printStackTrace();
                              }
                    }
                    output.add(xmltuples);
                    return output;
          }
}
```

A.6 ChangeLOC

```
import java.io.IOException;
import java.util.Iterator;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.DataBag;
import org.apache.pig.data.DefaultTupleFactory;
import org.apache.pig.data.Tuple;

```

```
/**
 * @author Ian Shang
 * @version 1.0
 * evaluation UDF to generate the changed LOC of every commit
 */
public class ChangeLOC extends EvalFunc<Tuple> {

        /**
         * @author Ian Shang
         * @version 1.0
         * @param Tuple with a databag of commits.
         * @return Tuple with changed LOC of every commit.
         */
        public Tuple exec(Tuple input) throws IOException {
                Iterator<Tuple> databag = ((DataBag) input.get(0)).
                    iterator();
                Tuple output = DefaultTupleFactory.getInstance().
                    newTuple();
                Tuple tname = databag.next();
                output.append((String) tname.get(0));
                int count = 0;
                while (databag.hasNext()) {//for every commit
                        Tuple commit = databag.next();
                        String add = (String) commit.get(5);//added
                            LOC of this commit
                        String del = (String) commit.get(6);//
                            deleted LOC of this commit

                        //convert String format of LOC to integer
                        int addi = 0;
                        if (add != null && add != "")
                                addi = Integer.parseInt(add);
                        int deli = 0;
                        if (del != null && add != "")
                                deli = Integer.parseInt(del);
                        if (addi > deli)
                                count += addi;
                        else
                                count += deli;
                }
                output.append(count);//accumulate the LOC
                return output;
        }
}
```

A.7 EvoAnalysisComment

```
import java.io.IOException;
import java.io.StringReader;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.DataBag;
import org.apache.pig.data.DefaultTupleFactory;
import org.apache.pig.data.Tuple;
import org.w3c.dom.Document;
import org.xml.sax.InputSource;
import JrexUtil.ChangeUnit;
import JrexUtil.ParseCvs;

/**
 * @author Ian Shang
 * @version 1.0
 * evaluation UDF to perform evolutionary analysis for only comment
    data.
 */
public class EvoAnalysisComment extends EvalFunc<Tuple> {
        String filename = "";

        /**
         * @author Ian Shang
         * @version 1.0
         * @param Tuple
         *              with a databag of every snapshot of XML
            representation of Java
         *              source code.
         * @return Tuple with evolutionary results of comments of
            the Java source
         *          code file.
         */
        public Tuple exec(Tuple input) throws IOException {

            Tuple returntuple = DefaultTupleFactory.getInstance
                ().newTuple();
            Iterator<Tuple> databag = ((DataBag) input.get(0)).
                iterator();

            Tuple tname = (Tuple) databag.next();
            filename = (String) tname.get(0);

            Tuple changeunittuples = DefaultTupleFactory.
                getInstance().newTuple();
            Tuple commits = databag.next();

            //get xml data of every commit
            List<String[]> xmlList = new ArrayList<String[]>();

            for (int i = 0; i < commits.size(); i++) {
                    Tuple commit = (Tuple) commits.get(i);
                    String version = (String) commit.get(0);
                    String xmlcode = (String) commit.get(1);
                    String[] temp = { version, xmlcode };
                    xmlList.add(temp);
            }
            for (int i = xmlList.size() - 1; i > 0; i--) {
                    String late = xmlList.get(i)[1];
                    String newone = xmlList.get(i - 1)[1];
                    Document latedoc = null;
                    Document newdoc = null;

                    try {
```

```
                    DocumentBuilderFactory
                        docbuilderfactory =
                        DocumentBuilderFactory
                                .newInstance();
                    DocumentBuilder docbuilder =
                        docbuilderfactory
                                .newDocumentBuilder
                                    ();
                    InputSource source = new InputSource
                        (new StringReader(late));
                    latedoc = docbuilder.parse(source);
            } catch (Exception e) {
                    System.out.println(e.getMessage());
            }
            try {
                    DocumentBuilderFactory
                        docbuilderfactory =
                        DocumentBuilderFactory
                                .newInstance();
                    DocumentBuilder docbuilder =
                        docbuilderfactory
                                .newDocumentBuilder
                                    ();
                    InputSource source = new InputSource
                        (new StringReader(newone));
                    newdoc = docbuilder.parse(source);

            } catch (Exception e) {
                    System.out.println(e.getMessage());
```

```
                                }
                                //use J-REX library to diff the two xml data
                                List<ChangeUnit> changes = ParseCvs.diff(
                                    newdoc, latedoc);
                                List<String[]> changeunitlist =
                                    CUtoStringArrayAndFilter(changes);

                                if (changeunitlist != null && changeunitlist
                                    .size() != 0) {

                                          for (String[] changeunit :
                                              changeunitlist) {
                                                Tuple cutuple =
                                                    DefaultTupleFactory.
                                                    getInstance()
                                                                    .newTuple();

                                                cutuple.append(xmlList.get(i
                                                     - 1)[0]);
                                                for (String s : changeunit)
                                                    {
                                                          cutuple.append(s);
                                                }
                                                changeunittuples.append(
                                                    cutuple);
                                          }
                                }

                      }
```

```
                if (changeunittuples.size() != 0) {
                        returntuple.append(filename);
                        returntuple.append(changeunittuples);
                }
                return returntuple;

        }
        /**
         * @author Ian Shang
         * @version 1.0
         * @param a list of changed code unit
         * @return turn change unit to string array and only keep
             comment changes
         */
        private List<String[]> CUtoStringArrayAndFilter(List<
            ChangeUnit> culist) {
                List<String[]> returnlist = new ArrayList<String
                    []>();
                for (ChangeUnit cu : culist) {
                        if (cu.getEntityType().toLowerCase().
                            contains("comment")) {
                                String[] changeunit = {
                                cu.getEntity().replace("#document()/
                                    File(" + filename + ")", ""),
                                cu.getEntityType(),
                                cu.getChangeType() };
                                returnlist.add(changeunit);
                        }
                        returnlist.addAll(CUtoStringArrayAndFilter(
                            cu.getChildchanges()));
                }
                return returnlist;
        }
}
```

A.8 EvoAnalysisMethod

```
import java.io.IOException;
import java.io.StringReader;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.DataBag;
import org.apache.pig.data.DefaultTupleFactory;
import org.apache.pig.data.Tuple;
import org.w3c.dom.Document;
import org.xml.sax.InputSource;
import JrexUtil.ChangeUnit;
import JrexUtil.ParseCvs;

/**
 * @author Ian Shang
 * @version 1.0
 * evaluation UDF to perform evolutionary analysis for only method
     data.
 */
public class EvoAnalysisMethod extends EvalFunc <Tuple>
{
        String filename="";
        /**
         * @author Ian Shang
         * @version 1.0
         * @param Tuple
         *            with a databag of every snapshot of XML
              representation of Java
         *            source code.
         * @return Tuple with evolutionary results of methods of the
               Java source
         *         code file.
         */
        public Tuple exec(Tuple input) throws IOException {
                Tuple returntuple = DefaultTupleFactory.getInstance
                    ().newTuple();
                Iterator<Tuple> databag = ((DataBag) input.get(0)).
                    iterator();

                Tuple tname = (Tuple) databag.next();
                filename = (String) tname.get(0);
                Tuple changeunittuples = DefaultTupleFactory.
                    getInstance().newTuple();
                Tuple commits = databag.next();
                // get xml data of every commit
                List<String[]> xmlList = new ArrayList<String[]>();
                for (int i = 0; i < commits.size(); i++) {
                        Tuple commit = (Tuple) commits.get(i);
                        String version = (String) commit.get(0);
                        String xmlcode = (String) commit.get(1);
                        String[] temp = { version, xmlcode };
                        xmlList.add(temp);
                }
                for (int i = xmlList.size() - 1; i > 0; i--) {
                        String late = xmlList.get(i)[1];
                        String newone = xmlList.get(i - 1)[1];
                        Document latedoc = null;
                        Document newdoc = null;
                        try {
                                DocumentBuilderFactory
                                    docbuilderfactory =
                                    DocumentBuilderFactory
                                                .newInstance();
                                DocumentBuilder docbuilder =
                                    docbuilderfactory
                                                .newDocumentBuilder
                                                    ();
                                InputSource source = new InputSource
                                    (new StringReader(late));
                                latedoc = docbuilder.parse(source);
                        } catch (Exception e) {
                                System.out.println(e.getMessage());
                        }
                        try {
```

```
                                        DocumentBuilderFactory
                                            docbuilderfactory =
                                            DocumentBuilderFactory
                                                           .newInstance();
                                        DocumentBuilder docbuilder =
                                            docbuilderfactory
                                                           .newDocumentBuilder
                                                              ();
                                        InputSource source = new InputSource
                                            (new StringReader(newone));
                                        newdoc = docbuilder.parse(source);

                              } catch (Exception e) {
                                        System.out.println(e.getMessage());
                              }
                              // use J-REX library to diff the two xml
                                  data
                              List<ChangeUnit> changes = ParseCvs.diff(
                                  newdoc, latedoc);
                              List<String[]> changeunitlist =
                                  CUtoStringArrayAndFilter(changes);

                              if (changeunitlist != null && changeunitlist
                                  .size() != 0) {

                                        for (String[] changeunit :
                                            changeunitlist) {
                                                Tuple cutuple =
                                                    DefaultTupleFactory.
                                                    getInstance()
                                                                  .newTuple();

                                                cutuple.append(xmlList.get(i
                                                     - 1)[0]);
                                                for (String s : changeunit)
                                                    {
                                                        cutuple.append(s);
                                                }
                                                changeunittuples.append(
                                                    cutuple);
                                        }
                                }

                        }
                        if (changeunittuples.size() != 0) {
                                returntuple.append(filename);
                                returntuple.append(changeunittuples);
                        }
                        return returntuple;

        }
        /**
         * @author Ian Shang
         * @version 1.0
         * @param a list of changed code unit
```

```
     * @return turn change unit to string array and only keep
        method changes
     */
    private List<String[]> CUtoStringArrayAndFilter(List<
        ChangeUnit> culist) {
            List<String[]> returnlist = new ArrayList<String
                []>();
            for (ChangeUnit cu : culist) {
                    if (cu.getEntityType().toLowerCase().
                        contains("method")) {
                            String[] changeunit = {
                            cu.getEntity().replace("#document()/
                                File(" + filename + ")", ""),
                            cu.getEntityType(),
                            cu.getChangeType() };
                            returnlist.add(changeunit);
                    }
                    returnlist.addAll(CUtoStringArrayAndFilter(
                        cu.getChildchanges()));
            }
            return returnlist;
    }
}
```

A.9 GetMethod

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.DataBag;
import org.apache.pig.data.DefaultBagFactory;
import org.apache.pig.data.DefaultTupleFactory;
import org.apache.pig.data.Tuple;
import JrexUtil.JavaFile;
import JrexUtil.ClassUnit;
import JrexUtil.Method;

/**
 * @author Ian Shang
 * @version 1.0
 * evaluation UDF to extract method from source code.
 */
public class GetMethod extends EvalFunc<DataBag> {
        /**
         * @author Ian Shang
         * @version 1.0
         * @param Tuple with snapshots of a Java source code file.
         * @return Databag with methods content.
         */
        public DataBag exec(Tuple input) throws IOException {
                Iterator<Tuple> databag = ((DataBag) input.get(0)).
                    iterator();
                Tuple sub = null;
                List<Object> sublist = null;
                if (input.size() > 1) {
```

```
                sub = (Tuple) input.get(1);
                sublist = sub.getAll();
            }
            Tuple tname = (Tuple) databag.next();
            String filename = (String) tname.get(0);

            DataBag output = DefaultBagFactory.getInstance().
               newDefaultBag();
            output.add(tname);
            //get java file snapshots
            Tuple javafiletuples = DefaultTupleFactory.
               getInstance().newTuple();
            List<Tuple> templist = new ArrayList<Tuple>();

            while (databag.hasNext()) {
                templist.add(databag.next());
            }
            //for every commit
            for (int i = 0; i < templist.size(); i++) {
                Tuple commit = templist.get(i);
                String sourcecode = (String) commit.get(1);
                String startdate = (String) commit.get(3);
                String enddate = startdate;
                if (i != templist.size() - 1)
                    enddate = (String) templist.get(i +
                       1).get(3);
                JavaFile jf = JrexUtil.ParseJava.
                   parseJavaCode(sourcecode,
                        filename, sublist);
                    if (jf == null) {
                            return null;
                    }
                    //for every class unit in the java file
                    for (ClassUnit cu : jf.getClassUnits()) {
                            for (Method m : cu.getMethodList())
                               {
                                    Tuple methodtuples =
                                       DefaultTupleFactory.
                                       getInstance()
                                                        .newTuple();

                                    methodtuples.append(m.
                                       getName());
                                    methodtuples.append(m.
                                       getModifier());
                                    methodtuples.append(m.
                                       getReturnType());
                                    methodtuples.append(
                                       startdate);
                                    methodtuples.append(enddate)
                                       ;
                                    Tuple paratuples =
                                       DefaultTupleFactory.
                                       getInstance()
                                                        .newTuple();
                                    for (String[] parameter : m.
                                       getParameterList()) {
```

```
                                                        paratuples.append(
                                                            parameter[0]);
                                                }
                                                methodtuples.append(
                                                    paratuples);
                                                methodtuples.append(m.
                                                    getContent());
                                        }
                                }

                        }
                        output.add(javafiletuples);
                        return output;
                }
}
```

A.10 CloneDetection

```
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.DefaultTupleFactory;
import org.apache.pig.data.Tuple;

/**
 * @author Ian Shang
 * @version 1.0
 * evaluation UDF to detect clone by using CC-Finder. This UDF
     relies on the external CC-Finder tool.
 */
public class CloneDetection extends EvalFunc <Tuple>
{
        /**
         * @author Ian Shang
         * @version 1.0
         * @param Tuple with a databag of method pairs.
         * @return Tuple with clone detection result by CC-Finder.
         */
        public Tuple exec(Tuple input) throws IOException {
                Tuple methodpair = (Tuple) input.get(1);
                Tuple methodA = (Tuple) methodpair.get(0);
                Tuple methodB = (Tuple) methodpair.get(1);

                // get method contents
                String contentA = (String) methodA.get(6);
                String contentB = (String) methodB.get(6);

                ProcessBuilder pb = null;
                // save content to temp file
                File a = new File("tempA");
                File b = new File("tempB");
                a.createNewFile();
                b.createNewFile();
                FileWriter fwa = new FileWriter(a);
                fwa.append(contentA);
                fwa.flush();
                fwa.close();

                FileWriter fwb = new FileWriter(b);
                fwb.append(contentB);
                fwb.flush();
                fwb.close();

                //use program wrapper to call CC-Finder
                pb = new ProcessBuilder("./ccfx", "d", "cpp", "-d␣
                    tempA␣", "-is",
                                "-d␣tempB", "-w", "f-w-g+", "-k", "
                                    1024M", "-b", "150", "-o",
                                "tempresult.ccfxd");
                //save result to a temp file
                File result = new File("tempresult.ccfxd");
                //read the output of clone detection from the temp
                    file
                FileReader fr = new FileReader(result);
                String logStr = "";
                char[] readbuffer = new char[51200];
                while ((fr.read(readbuffer)) > 0) {
                        logStr = logStr.concat(new String(readbuffer
                            ).replaceAll("\0", ""));
                        readbuffer = new char[51200];
                }
                readbuffer = null;
                fr.close();
```

```
                    Tuple returntuple = DefaultTupleFactory.getInstance
                        ().newTuple();
                    returntuple.append(input.get(0));
                    returntuple.append(logStr);
                    return returntuple;
          }
}
```

A.11 TimeOverlap

```
import java.io.IOException;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import org.apache.pig.FilterFunc;
import org.apache.pig.data.Tuple;

/**
 * @author Ian Shang
 * @version 1.0
 * filter UDF to check if two methods have time overlap.
 */
public class TimeOverlap extends FilterFunc {
          /**
           * @author Ian Shang
           * @version 1.0
           * @param Tuple with two methods.
```

```
     * @return boolean value to indicate if two methods have
        time overlap.
     */
    public Boolean exec(Tuple input) throws IOException {
            //get two method
            Tuple methodpair = (Tuple) input.get(1);
            Tuple methodA = (Tuple) methodpair.get(0);
            Tuple methodB = (Tuple) methodpair.get(1);
            //get the time span of two methods
            String startA = (String) methodA.get(3);
            String endA = (String) methodA.get(4);

            String startB = (String) methodB.get(3);
            String endB = (String) methodB.get(4);
            //parse date time
            SimpleDateFormat dateparser = new SimpleDateFormat("
                yyyy/MM/dd");
            Date startdateA = new Date();
            Date enddateA = new Date();
            try {
                    startdateA = dateparser.parse(startA);
                    enddateA = dateparser.parse(endA);
            } catch (ParseException e) {
                    e.printStackTrace();
            }

            Date startdateB = new Date();
            Date enddateB = new Date();
            try {
```

```
                        startdateB = dateparser.parse(startB);
                        enddateB = dateparser.parse(endB);
                } catch (ParseException e) {
                        e.printStackTrace();
                }
                if (startdateA.after(enddateB) || startdateB.after(
                    enddateA))
                        return false;
                else
                        return true;
        }
}
```

A.12 TimeSpan

```
import java.io.IOException;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Iterator;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.DataBag;
import org.apache.pig.data.DefaultBagFactory;
import org.apache.pig.data.Tuple;

/**
 * @author Ian Shang
 * @version 1.0
 * evaluation UDF to to group commits by time span.
 */
public class TimeSpan extends EvalFunc<DataBag> {
        /**
         * @author Ian Shang
         * @version 1.0
         * @param Tuple with commits.
         * @return Databag with commits, which are given time span
            as key value.
         */
        public DataBag exec(Tuple input) throws IOException {
                Iterator<Tuple> databag = ((DataBag) input.get(0)).
                    iterator();

                Tuple tname = databag.next();
                DataBag output = DefaultBagFactory.getInstance().
                    newDefaultBag();
                output.add(tname);
                int timespansize=((Tuple)input.get(1)).size();
                //get the time span
                String [] timespans= new String[timespansize];
                for(int i=0;i<timespansize;i++)
                {
                        timespans[i]=(String) ((Tuple) input.get(1))
                            .get(i);
                }

                while (databag.hasNext()) {
                        Tuple commit = databag.next();
                        String timestr = (String) commit.get(3);
                        SimpleDateFormat formatDate = new
                            SimpleDateFormat(
                                        "yyyy/MM/dd hh:mm:ss");
                        Date date = new Date();
                        try {
                                date = formatDate.parse(timestr);
                        } catch (ParseException e) {
                                e.printStackTrace();
                        }
                        //check time span

                        SimpleDateFormat dateparser = new
                            SimpleDateFormat("yyyy/MM/dd");
                        //for every time span, check this change is
                            in which one
                        for(int i=0;i<timespansize-1;i++)
                        {
                                Date startdate = new Date();
                                Date enddate = new Date();
                                try {
                                        startdate = dateparser.parse
                                            (timespans[i]);
                                        enddate = dateparser.parse(
                                            timespans[i+1]);
                                } catch (ParseException e) {
                                        e.printStackTrace();
                                }
```

```
                                        //if find a corresponding time span,
                                             use the date to be the key to be
                                             grouped
                                        if (date.before(enddate) && date.
                                            after(startdate)) {
                                                commit.append(timespans[i]+"
                                                    -"+timespans[+1]);
                                                output.add(commit);
                                                break;
                                        }
                              }
                    }
                    return output;
          }
}
```

A.13 IsBug

```
import java.io.IOException;
import org.apache.pig.FilterFunc;
import org.apache.pig.data.Tuple;

/**
 * @author Ian Shang
 * @version 1.0
 * filter UDF to check if a change indicates a bug.
 */
public class IsBug extends FilterFunc {
    /**
     * @author Ian Shang
     * @version 1.0
     * @param Tuple with commits.
     * @return boolean value to indicate if a commit indicates
         bug.
     */
    public Boolean exec(Tuple input) throws IOException {

            Tuple commit = input;
            String comment = (String) commit.get(7);
            String type = "UNKNOWN";

            comment = comment.toLowerCase();
            //use commit log to check
            if (comment.contains("copyright")) {
                    type = "COPYRIGHT";
            } else if (comment.contains("HEAD") || comment.
                contains("sync")
                                || comment.contains("fsync") ||
                                    comment.contains("current")
                                || comment.contains("branch") ||
                                    comment.contains("merge")
                                || comment.contains("changes") ||
                                    comment.contains("pull_up")
                                || comment.contains("rev") ||
                                    comment.contains("bring")
                                || comment.contains("changes")) {
                    type = "BRANCH_SYNC";
                } else if (comment.contains("fix") || comment.
                    contains("bug")
                                || comment.contains("repair") ||
                                    comment.contains("problem")
                                || comment.contains("crash") ||
                                    comment.contains("eliminate")) {
                        type = "BUG";
                } else if (comment.contains("add")) {
                        type = "NEW";
                } else if (comment.contains("indent") || comment.
                    contains("RCS")
                                || comment.contains("Wall") ||
                                    comment.contains("clean up")
                                || comment.contains("clean up")) {
                        type = "INDENT";
                }

                if (type == "BUG")
                        return true;
                else
                        return false;
        }
}
```

A.14 IsFI

```
import java.io.IOException;
import org.apache.pig.FilterFunc;
import org.apache.pig.data.Tuple;

/**
 * @author Ian Shang
 * @version 1.0
 * filter UDF to check if a change introduces new features.
 */
public class IsFI extends FilterFunc {
        /**
         * @author Ian Shang
         * @version 1.0
         * @param Tuple with commits.
         * @return boolean value to indicate if a commit introduces
             new features.
         */
        public Boolean exec(Tuple input) throws IOException {

                Tuple commit = input;
                String comment = (String) commit.get(7);
                String type = "UNKNOWN";
                comment = comment.toLowerCase();
                //use commit log to check
                if (comment.contains("copyright")) {
                        type = "COPYRIGHT";
                } else if (comment.contains("HEAD") || comment.
                    contains("sync")
                                || comment.contains("fsync") ||
                                   comment.contains("current")
```

```
                                        || comment.contains("branch") ||
                                           comment.contains("merge")
                                        || comment.contains("changes") ||
                                           comment.contains("pull␣up")
                                        || comment.contains("rev") ||
                                           comment.contains("bring")
                                        || comment.contains("changes")) {
                        type = "BRANCH_SYNC";
                } else if (comment.contains("fix") || comment.
                    contains("bug")
                                        || comment.contains("repair") ||
                                           comment.contains("problem")
                                        || comment.contains("crash") ||
                                           comment.contains("eliminate")) {
                        type = "BUG";
                } else if (comment.contains("add")) {
                        type = "NEW";
                } else if (comment.contains("indent") || comment.
                    contains("RCS")
                                        || comment.contains("Wall") ||
                                           comment.contains("clean␣up")
                                        || comment.contains("clean␣up")) {
                        type = "INDENT";
                }

                if (type == "NEW")
                        return true;
                else
                        return false;
```

```
        }
}
```