Using Pig as a Data Preparation Language for Large-Scale Mining Software Repositories Studies: An Experience Report

Weiyi Shang, Bram Adams, Ahmed E. Hassan

Software Analysis and Intelligence Lab (SAIL), Queen's University, Kingston, Canada, K7L 3N6

Abstract

The Mining Software Repositories (MSR) field analyzes software repository data to uncover knowledge and assist development of ever growing, complex systems. However, existing approaches and platforms for MSR analysis face many challenges when performing large-scale MSR studies. Such approaches and platforms rarely scale easily out of the box. Instead, they often require custom scaling tricks and designs that are costly to maintain and that are not reusable for other types of analysis. We believe that the web community has faced many of these software engineering scaling challenges before, as web analyses have to cope with the enormous growth of web data. In this paper, we report on our experience in using a web-scale platform (i.e., Pig) as a data preparation language to aid large-scale MSR studies. Through three case studies, we carefully validate the use of this web platform to prepare (i.e., Extract, Transform, and Load, ETL) data for further analysis. Despite several limitations, we still encourage MSR researchers to leverage Pig in their large-scale studies because of Pig's scalability and flexibility. Our experience report will help other researchers who want to scale their analyses.

Key words: Software engineering; Mining Software Repositories; Pig; MapReduce

1. Introduction

Software projects and systems continue to grow in size and complexity. The first version of the *Linux kernel*, which was released in 1991, consisted of 10,239 lines of source code (SLOC), while version *2.6.32* released in 2009 consists of 12,606,910 SLOC. In eighteen years, the size of the Linux kernel has increased by a factor larger than 1,200. Similarly, Gonzalez-Barahona *et al.* find that the

Email address: {`swy, bram, ahmed`}`@cs.queensu.ca` (Weiyi Shang, Bram Adams, Ahmed E. Hassan)

size of the Debian Linux distribution doubles approximately every two years (25 mSLOC in 1998 vs. 288 mSLOC in 2007) [1, 2]. Moreover, recent work by Mockus shows that a universal repository of the version history of all open source software systems available online contains TBs of data and that the process to collect such a repository is rather lengthy and complicated, taking over a year [3]. The size of the code available 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 [4]. 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 to uncover knowledge and assist software development. The process adopted by most large-scale MSR studies is similar to the Extract - Transform - Load (ETL) data preparation process used by data warehouses, i.e., data is extracted from software repositories, transformed into certain formats and loaded into a data warehouse. MSR data preparation is typically performed by specialized programming scripts in general-purpose query or scripting languages. The prepared data is then further analyzed using modelling tools like R [5], Weka [6], or other specially built tools.

As the size of software repository data increases, more complex platforms are needed to enable rapid and efficient ETL of data. Software engineering researchers typically try to scale up ETL by means of specialized one-off solutions that are hard to reuse and that are costly to maintain. For example, to identify clones across the FreeBSD operating system (131,688 kSLOC), Livieri *et al.* developed their own distributed platform dedicated to performing large-scale code clone detection [7]. We believe that in many ways the ETL phase is seen as a necessary evil by most researchers. Having to tweak tools to scale up to bigger data may distract focus and effort from the actual MSR analysis on the prepared data.

Many of the challenges associated with data preparation in MSR studies have already been faced in the web field by companies like Google and Facebook. The web field has developed several platforms to enable the large-scale preparation and processing of web-scale data sets, for example to analyze web crawl data or to process personal messages. Hadoop [8] and Pig [9] are examples of such platforms. We firmly believe that the software engineering field can adopt many of these platforms to scale MSR studies. For instance, in prior work [10, 11] we showed that we could scale and speed-up software evolution studies using Hadoop, an open-source implementation of MapReduce, which is a distributed framework based on a simple programming model [12].

However, our prior experience highlighted some of the limitations of Hadoop as a platform for large-scale software engineering analysis. In particular, the use of Hadoop requires rather tedious low-level programming effort. To counter this limitation, this paper evaluates the use of a higher-level web-scale platform called Pig. Pig uses a high-level data processing language on top of Hadoop that sacrifices some scalability for increased flexibility. This paper evaluates Pig's ability of preparing data for large-scale MSR study.

Three case studies with Pig show that it can successfully prepare data for large-scale MSR studies, with similar scalability as Hadoop. The major contributions of this paper are:

1. We evaluate Pig's ability to prepare data in a modular way by performing three large-scale MSR studies in detail. Our implementation[1] can be reused by other MSR researchers.
2. We compare the use of Pig and Hadoop for preparing data for MSR studies.
3. We report the lessons learnt with Pig in order to assist other researchers who want to use Pig as a data preparation language in their MSR studies.

Our experience shows that, compared to existing MSR data preparation approaches, Pig has several advantages, such as scalability, modular design and flexible data schemas. However, there are also several disadvantages of using Pig, including extra programming effort and lack of debugging support. Despite such limitations, we believe that Pig can be adopted by other MSR researchers to assist in large-scale MSR data preparation.

The rest of the paper is organized as follows. Section 2 illustrates the data preparation of MSR studies.

Then, we present Pig by using it to prepare data for a simple MSR study in Section 3. We perform three detailed MSR studies with Pig to evaluate its ability to prepare data for large-scale MSR studies (Section 4). We evaluate Pig's performance in data preparation for large-scale MSR studies in Section 5 and distill the lessons we learned from using Pig as a data preparation language for MSR studies in Section 6. Finally, Section 8 discusses related work and Section 9 presents the conclusions of this paper.

2. Data preparation in MSR studies

In this section, we use a motivating example from our real-life research experience [14] to illustrate the data preparation (ETL) of MSR studies.

The data preparation tool that we use in this example is J-REX, i.e., a highly optimized ETL tool for Java systems similar to C-REX [15], with the following functionalities:

- Extracting every Java file revision from a CVS repository.
- Transforming Java source code into an XML representation.
- Abstracting line-level change information ("line 10 has changed") to the program entity level ("function f1 no longer calls function f2").
- Calculating software metrics, such as number of lines of code (#LOC).

[1]The source code is available in the first author's Master's thesis [13].

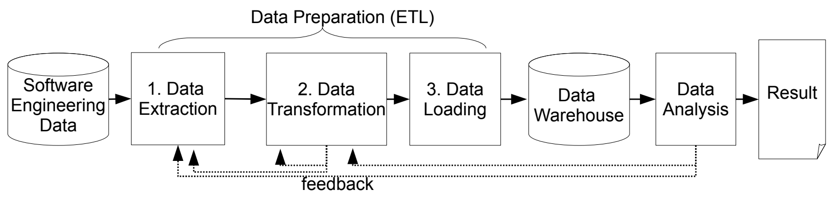

Figure 1: ETL pipeline for large-scale MSR studies.

Researcher Lily wants to study software defects and code clones using the source control repository of a large, long-lived software project.
Step 0. Initially, Lily tries to use evolutionary data prepared by the original J-REX [10]. This data describes the individual program entities that changed in each revision. However, since Lily wants to study code clones, she also needs the complete source code snapshots to perform code clone detection. Hence, the authors need to enhance the existing J-REX tool to support Lily's study.
Step 1. Lily requests every snapshot of each source code file and also a report about which methods were added or deleted in each snapshot. The authors modify the original J-REX to prepare the required data.
Step 2. In order to study software defects in the data that we prepare for her, Lily wants to use a heuristic that relates changes in source code to bugs by checking for keywords like "bug" or "fix" in the commit logs of the source control system. Since J-REX did not yet extract commit log data, the authors need to add this functionality to J-REX, as well as the heuristic for relating source code changes to bugs.
Step 3. While performing data analysis on the prepared data, Lily finds that the source code content of some methods is missing. She also requires the deletion of methods to be associated with the first snapshot after the deletion instead of with the snapshot of the deletion. For example, if method *foo* was in snapshot *1.0* but not in snapshot *1.1*, Lily needs the deletion of *foo* to be recorded for snapshot *1.1*, not for snapshot *1.0*.
Step 4. Because of the large-scale data, the authors spend much effort to fix bugs, prepare the new data, and deliver the prepared data to Lily. She makes great progress in her research, but now she has to perform clone detection on the extracted methods in the source code. As the authors are not clone detection experts, they choose to use an existing clone detection tool. Even though the authors already extracted the source code of methods, they now have to output the source code in the data format used by the clone detection tool. After doing the clone detection, the authors then need to collect the results and indicate if a method in the method list contains code clones.

As illustrated by the five steps above, MSR researchers need to perform multiple iterations of different types of analyses on prepared data from software repositories. Figure 1 summarizes the typical ETL pipeline of MSR studies, consisting of the following 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 or software archives. For example, bug repositories track the histories of bug reports and feature requests. Tools to process such repositories are typically implemented in general-purpose programming languages. In the motivating example, the data extraction uses J-REX to extract the source code change information and commit logs for the source control repository.

2. **Data transformation.** After the raw data is extracted from the software repositories and software archives, it typically needs to be abstracted and merged with other extracted data for further analysis.
   We performed a number of different data transformations in the motivating example in Section 2. For example, we transformed the extracted source code to a list of methods, we transformed the extracted commit log data to boolean values that indicate whether changes are related to bugs, and we processed the list of methods to a list of boolean values that indicate whether methods contain cloned code.
3. **Data loading.** In this phase, the transformed data is converted into the right format to be loaded into various types of analysis environments, such as relational databases, R and Weka, for further analysis. Data loading can even be as simple as writing transformed results to files in a certain format. In our motivating example, the output data is loaded into XML files that are prepared for further analysis.

Data preparation (ETL) is a highly iterative 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, as our example made clear.

This paper focuses on the data preparation (ETL) steps in Figure 1. We want to improve the iterative process of MSR data preparation by making it modular and scalable. We use a distributed framework from the web community, i.e., Pig, to achieve this. In the next section, we present the main concepts of Pig by using it to prepare data for a simple MSR study.

3. PIG

Pig [9] is a Hadoop-based [8] platform designed for analyzing massive amounts of data. Pig provides a high-level data processing language called Pig Latin [9]. To illustrate how Pig can prepare data for MSR studies, we use it in a simple MSR study to measure the evolution of #LOC (number of LOC) in the different snapshots of the source code in a software project. The corresponding Pig Latin code is shown in Figure 2. All variables in our code snippets 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, line 1 loads all data from a CVS repository as a (*"file name"*, *"file content"*) pair into Pig storage, This storage

```
1 RAWDATA = load '$inputdata' 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: Pig Latin script for measuring the evolution of the total number of lines of code (#LOC) in the different snapshots of a source control repository.

is based on the Hadoop Distributed File System [8] (HDFS), which conceptually stores data into different fields of a table, accessible by name or by field index. The value of the parameter *inputdata* is specified from the command line or specified by a parameter file.

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 3. In the Java source code shown in Figure 3, 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*. Using program wrappers in Java source code is the only way to access existing tools from Pig if their source code is not available. 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, line 3 parses every source code file's log data and generates the historical revision numbers of every source code file. The *"$0"* in line 3 represents the first field in the tuples of "HISTORYLOG". After generating the revision 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. Intermediate data of each step is accessible as variables, such as *CODE*, which can be examined during the process of analysis.

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"*.

To scale to large input data, Pig exploits another distributed framework from the web community called Hadoop. Hadoop is an implementation of the MapReduce programming paradigm [12]. MapReduce consists of two phases:

```
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␣", e);
        }
}
```

Figure 3: Java source code of the *exec* method of the programming unit *"ExtractLog"* (generating source code history log).

a massively parallel “Map” phase, followed by an aggregating “Reduce” phase. The input data for MapReduce is 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. Each group of intermediate data having the same key is then passed to a specific Reducer (processes assigned to the “Reduce phase”). Each Reducer performs computations on a group of data and reduces it to one single key/value pair. The output of all Reducers is the final result of the MapReduce process.

The Pig Latin script in Figure 2 will be transformed to Hadoop Java code that follows the MapReduce paradigm. For example, a possible transformation of the script in Figure 2 might consist of two steps of MapReduce:

1. A list of file data is extracted from the source control repository, containing the raw data of the history of each file. Each Mapper accepts a file as input, uses *rlog* to analyze it, collects the output of *rlog* and generates key/value pairs of the form (*“file name”*, *“rlog output of the file”*). Reducers accept these pairs and generate the revisions of each file depending on the *rlog* output. The output of the Reducers is represented as a key/value pair of the form (*“file name”*, *“revision number”*). If file “a.java” has two revisions 1.0 and 2.0, the output contains two key/value pairs (*“a.java”*, *“1.0”*) and (*“a.java”*, *“2.0”*).
2. Mappers take the output pairs of the previous step to extract the specific code revision given by the pair’s value, count the #LOC of the correspond-

ing source file and generate intermediate key/value pairs of the form (*"revision number"*, *"#LOC"*). For example, for a file named "a.java" with 100 LOC in revision *1.0*, a Mapper would generate a key/value pair of (*"1.0"*, *"100"*). Afterwards, each list of key/value pairs with the same key, i.e., revision number, is sent to the same Reducer, which sums all #LOCs in the list, and generates output as a key/value pair of the form (*"revision number"*, *"SUM #LOC"*). If a Reducer receives a list with key *"1.0"*, and the list consists of two values *"100"* and *"200"*, the Reducer will sum the values *"100"* and *"200"* and output (*"1.0"*, *"300"*).

Without Pig, a researcher would need to manually implement the above MapReduce steps. Each step requires tedious, low-level programming effort, which is a burden for MSR researchers, especially because none of this low-level code is easily reusable. For example, in our previous work, we migrated J-REX to Hadoop [10]. In the Hadoop version of J-REX, more than 80 percent of the source code consisted of such boiler-plate Java code.

Pig's high-level preparation language is promising, since it requires significantly less programming effort than Hadoop. The Pig Java functions that need to be implemented to run existing MSR tools still represent some overhead; yet can easily be reused for other MSR analyses. In our experience, the portion of boiler-plate code in Pig is only around 40% to 50%, which is much lower than that of Hadoop. In the next section, we discuss three case studies in which we evaluate Pig as a data preparation language for large-scale MSR studies.

4. Experience report

In this section, we present our experience of preparing data for three large-scale MSR studies using Pig. We first explain the requirements and implementation of each case study in detail and then we evaluate the modular design of our Pig implementations.

4.1. Case study requirements and implementation

We use Pig to perform data preparation (ETL) on three MSR studies. For each study, we show what data is required for the analysis and how we implemented the data's preparation with Pig. The subject system for the three case studies is the source control system of *Eclipse*, which contains around 10GB of data.

Study one: correlation between comment updates and bugs

The first software 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 source code comments?

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

1. Loading data from a CVS repository into Pig storage as a (*"filename"*, *"file content"*) pair.
2. Generating log data for every source code file.
3. Generating a list of revision numbers and commit logs for every source code file.
4. Using heuristics on commit logs to check if a change contains a bug fix.
5. Extracting every revision of source code for every source code file.
6. Transforming every snapshot of every source code file into XML format.
7. Comparing every two consecutive revisions of source code to check whether there is any comment change.

```
 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 BUGCHANGES=filter HISTORYVERSIONS by IsBug{$0};
 5 NOBUGCHANGES=filter HISTORYVERSIONS by not IsBug{$0
       };
 6 CODE=foreach HISTORYVERSIONS generate
       ExtractSourceCode($0);
 7 XMLS=foreach CODE generate ConvertSourceToXML($0);
 8 COMMENTEVO=foreach XMLS generate EvoAnalysisComment
       ($0);
 9 BUGRESULT=join BUGCHANGES by $0.$0, COMMENTEVO by
       $0.$0;
10 NOBUGRESULT=join NOBUGCHANGES by $0.$0, COMMENTEVO
       by $0.$0;
11 dump BUGRESULT;
12 dump NOBUGRESULT;
```

Figure 4: Pig Latin script for study one.

The Pig Latin source code for study one is shown in Figure 4. In the script, 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 revisions from the CVS log data. Line 4 and line 5 check if a change is related to bugs. Line 6 extracts every historical revision of all the source code files. These snapshots of source code files are transformed to XML files by line 7. Line 8 analyzes the evolution of comments of every source code file. Line 9 and line 10 join the evolution of comments, i.e., the output of line 8, with the bug-related changes and non-bug-related changes respectively. The Pig Latin script expresses high-level information about the process of the MSR study, in contrast to the low-level of Hadoop programs. Pig helps researchers focus on the MSR study itself instead of on the implementation details.

Study two: correlation between code clones and bugs

The second software study is an empirical study on software defects in both cloned and non-cloned methods in a software system [14], which is actually the motivating example presented in Section 2.

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

1. is the revision a bug fix?
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 the program units in lines 1 to 7 from study one (Figure 4) without any modification. In addition, we also need program units for:

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

The Pig Latin script for study two, which re-uses the existing variables from study one, is shown in Figure 5.

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

Figure 5: Pig Latin script for study two.

In this Pig Latin script, line 1 analyzes the evolution of methods in every source code file. Line 2 generates the source code content of every method in each source code file. To perform clone detection, line 3 generates the cross product of all method contents. The cross products consist of pairs of method content, such that line 4 can perform clone detection between 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 5 filters out those false code clones. Lines 6 and 7 join the evolutionary data of methods, the result of code clone detection, and historical revisions respectively related and not related to bugs. Lines 8 and 9 dump the results to a terminal. Alternatively, the results can be stored into files by using keyword "store" instead of "dump". The Pig Latin language directly supports commonly used functionalities like joins, which substantially simplifies the implementation of MSR studies using Hadoop.

Study three: evolution of the complexity of source code changes

In the third experiment, we prepare data to calculate the evolution of the complexity of source code changes. Hassan uses this data to predict software defects [16].

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

Implementation:

Study three re-uses line 1 to 3 in study one (Figure 4). Three more program units are required:

1. Checking for every change whether the change is an FI change.
2. Grouping changes per time period. In particular, we focus on the four quarters in 2008.
3. Counting the changed #LOC.

The corresponding Pig Latin script, which uses the variables from experiment one and two, is shown in Figure 6. Line 2 uses five specific days to indicate the four quarters in 2008 as time spans and line 3 uses the key value generated by line 2 as *"$8"* to group the commits into time spans. Line 4 counts the changed #LOC of every group of changes generated by line 3.

The Pig scripts of the three MSR studies show that the high-level language of Pig Latin helps focus on the process of the MSR studies rather than on the details of implementation or parallelization. However, one still needs to implement Java modules that will be called by Pig scripts.

4.2. Modular design overview

Pig stimulates a modular design in which each Pig program is decomposed into a number of small program units. With such a modular programming style,

```
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 6: Pig Latin script for study three.

adding a new program unit or changing one program unit does not affect other program units as long as the order of the fields of the data did not change. This flexibility reduces coupling between modules but necessitates good documentation about the format of the data. Program units also comprise dedicated Java modules. These Java modules are automatically reused when the program unit that they belong to is used for data preparation of different MSR studies, which can be as simple as wrapping an existing tool in a separate process.

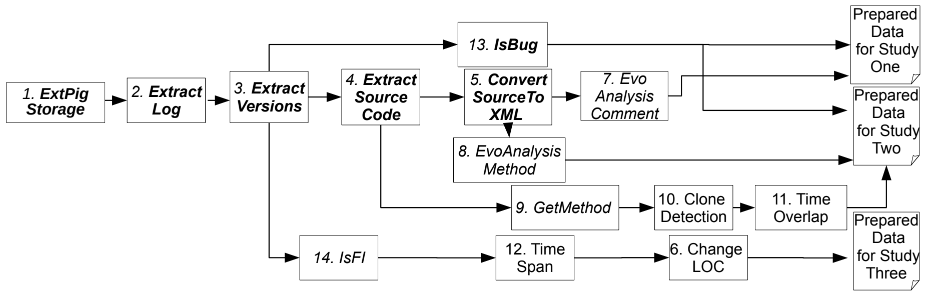

Figure 7: Composition of the data preparation process for the three MSR studies performed with PIG. Modules with name in bold are used by more than one case study, whereas modules with name in italic are used by J-REX.

The program units that we identified for the three software studies are summarized in Figure 7. Three out of fourteen program units are reused in all three case studies and six out of fourteen program units are reused in at least two case studies (bold in Figure 7). Figure 7 also shows how we composed different program units into the data preparation process of the three MSR studies. The most widely re-used program units provide basic functionalities of MSR studies. The case studies are representative of most tasks done in MSR [17].

To conclude, we made the following findings during the implementation of three MSR studies using Pig: 1) the modular design of Pig supports the reuse of program units; 2) existing MSR tools can be migrated to Pig in a straightforward way.

5. Comparing Pig and Hadoop

In our previous research, we verified the feasibility of using Hadoop to prepare data for software studies [10, 11] by evaluating the J-REX data preparation tool on the Hadoop platform. In this section, we use 10 out of the 14 program units of the three MSR studies (italic in Figure 7) to migrate J-REX to the Pig platform and compare the Hadoop-based J-REX and the Pig-based J-REX. We use J-REX to perform the comparison for two reasons: 1) we have both Hadoop-based and Pig-based implementations of J-REX and 2) J-REX is well understood by the authors from previous studies and covers most (10 out of 14) of the program units in this paper.

Table 1: Comparison of the Hadoop and Pig based J-REX source code.

	# Java LOC	**# Pig script LOC**	**# Modules**	**# Boiler-plate LOC (%)**
Hadoop	277	N/A	1	235 (85%)
Pig	761	13	10	342 (45%)

5.1. Source code

We first compare the source code of the Hadoop and Pig implementations. An overview of this data is shown in Table 1. In our previous research [10], only 277 lines of additional Java code were required to migrate J-REX to Hadoop. In our case studies, we used less than 10 lines of Pig script for combining program units and 761 lines of java code for the various J-REX modules.

From our experience, preparing MSR data using Pig requires more Java code than using Hadoop, for a number of reasons. The Pig implementation has more modules than Hadoop (10 in Pig versus 1 in Hadoop for our implementation). Each module repeats boiler-plate code, taking up 45% of all Pig Java code. Moreover, the modular design of Pig requires researchers to break down the data preparation process into reusable modules that can be combined together. Therefore, the researchers cannot treat the overall process as a black box, but rather need to understand the process and design the interaction between modules. Hence, each module needs additional boiler-plate code for loading intermediate results and storing the output for later modules. These requirements increase the burden of developing the data preparation program, as well as the amount of source code. We consider the design of Java code in Pig as one of its major disadvantages. That said, once the initial effort for implementing a module has been invested, subsequent tools likely can reuse modules, reducing future development effort.

5.2. Program design

Designing a Hadoop-based data preparation program requires designing five classes for each MapReduce step: a Mapper class, a Reducer class, an Inputformat class, an Outputformat class and a RecordReader class. In addition, a driver class is required to combine all the MapReduce steps together. Hence, not only the Map and Reduce classes require effort. Moreover, one often needs to design customized data types, which increases effort even more.

In principle, a Hadoop program with multiple MapReduce steps can have a similar modular design as a Pig program, supporting multiple modules with intermediate data for each MapReduce step. However, such a design is not optimal in practice. Each step of MapReduce requires additional design effort for the five classes mentioned above. Moreover, additional I/O for reading and saving data to the disk and un-negligible start-up phase overhead are introduced by each MapReduce step. Therefore, Hadoop users typically design a minimal number of MapReduce steps.

Pig requires less design effort than Hadoop. Researchers only need to design one Java class for each step of a Pig-based data preparation program when the internal operators of Pig, such as GROUP and JOIN, cannot complete the functionality. Two additional classes are needed for the whole Pig program when customized data loading and storing is used. Since Pig uses an internal data format (which is similar to JSON [18]), no additional design for input and output data format of each step or data serialization is required, such that Pig users can focus on the real data processing. The Pig platform automatically transforms, schedules and combines Pig program units into a minimal number of MapReduce steps to avoid the overhead from initializing MapReduce and dealing with intermediate data. Therefore, Pig users typically design the Pig program with a relatively larger number of modules than the number of MapReduce steps in Hadoop.

Comparing the design of Hadoop and Pig programs, we first find that although Pig's modular design brings additional boiler-plate Java code, it assists in isolating bugs when testing the program. Second, the Pig program units are easier to reuse than Hadoop program. Third, we find that Pig has as advantage a less constrained design. For example, users of Pig do not have to design their program following the MapReduce paradigm. On the other hand, Hadoop users can customize their programs more. For example, the internal data format may not be the most suitable one, and users may want to define their own data type as well as the identity and equality of the customized data type.

5.3. Performance

In addition to the source code and design characteristics, we also compare the performance of Pig for MSR studies to the performance of Hadoop and non-distributed programming. Our previous experiment shows that using Hadoop on a 4 machine-cluster reduces the computation time by 60-70% when analyzing the CVS [19] source control repository of the Eclipse project [10]. Pig uses a high-level language that is automatically compiled into Hadoop code, which means that it sacrifices some performance for the flexibility and ease of implementation. Here, we want to examine the relative scalability of Pig compared to Hadoop. The performance evaluation of Pig should include both the running time of MSR studies and the time spent on iterative analysis.

Table 2: 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 per server
Network	Gigabit	Gigabit
OS	Ubuntu 9.10	Ubuntu 9.10
Disk type	SSD	SATA

In the experiment, we analyze three pieces of input data with different sizes on the non-distributed J-REX, the Hadoop-based J-REX and the Pig-based J-REX. We first used the optimized, non-distributed J-REX to prepare data from three major sub-folders of the *Eclipse* CVS repository on a powerful server machine (see Table 2). 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 2. The performance results are shown in Table 3.

Table 3: Average running time across three runs of J-REX on a single machine, the Hadoop platform and the Pig platform.

Sub-folder	**Size**	**On single machine**	**On Hadoop platform**	**On Pig platform**
runtime	11MB	12 min	2 min	1.5 min
e4	219MB	164 min	20 min	23 min
pde	269MB	240 min	16 min	24 min

The average running time across three runs is shown in Table 3. The average running time of the Pig-based J-REX turned out to be slightly higher than the average running time of the Hadoop-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 seem 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 [20]. However, in our experiments, we found that Pig sometimes could be more efficient than Hadoop-based J-REX. This can be explained by the fact that additional I/O is introduced and the Hadoop-based J-REX stores intermediate data after each step, while the intermediate data in the Pig-based J-REX is only calculated and stored on demand.

As shown in the introductory example in Section 2, 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. In our previous research, running J-REX on the whole Eclipse CVS repository took 80 minutes in a 10-node cluster [11], which shows that the running time of one iteration is not trivial. With Pig, MSR researchers can store and load data into variables as desired, without having to design and implement data formats and additional methods, which reduces the overall time to prepare MSR data. Yet, it is very hard to measure the actual time spent in our case studies on iterative analysis, for a number of reasons. First, the total time encompasses the time needed to think about changing the MSR tool and to test the tool, which is very hard to measure. Second, the learning effect from our earlier experience with the stand-alone and Hadoop versions of J-REX makes time information unreliable.

Still, we can get an idea of the order of magnitude of difference between Pig and MapReduce. J-REX on a single machine and the Pig platform both required development in a similar high-level programming language, such that given their time difference in Table 3, iterative execution will be much faster for Pig.

6. Other lessons learnt

In this section, we distill the lessons learnt from our experience with the three case studies that are not included in the comparison between Hadoop and Pig in Section 5.

6.1. Data storage

As Pig runs on top of Hadoop, the input data of a Pig program needs to be loaded into Hadoop data storage, i.e., HDFS [8]. The data prepared by Pig also needs to be copied from HDFS to the local file system if researchers need to use other data analysis techniques like R [5] and Weka [6] to analyze the prepared data. Because software engineering data is typically large, the performance and efficiency of data storage is important for Pig to prepare data for MSR studies. We now discuss the advantages and disadvantages of the data storage of Pig.

The advantage of HDFS includes optimized data reading and fault tolerance. HDFS optimizes the performance of data reading [8], which takes up most of the I/O in MSR studies [17]. Hence, although data in software repositories is growing ever larger and faster, the scalability of Pig-based data preparation is not limited by I/O bandwidth. HDFS also provides a fault tolerance mechanism to ensure the correctness and completeness of software repository data, since machine failure is common in large distributed computing environments.

However, HDFS also brings disadvantages to the users of Pig. The most important disadvantage is that HDFS does not support either updating or appending data. This disadvantage prevents mining software data incrementally. In addition, as mentioned above, MSR data needs to be loaded into HDFS before being processed by Pig, which corresponds to run-time overhead.

6.2. Data structure

Pig Latin's default data structure is a flexible data format, as mentioned in Section 5. However, from our experience and the scripts in Figure 4, 5 and 6, we find that the data format of the input and output of program units in Pig

Latin is not explicitly specified. Users of Pig need to know the corresponding data format when they want to reuse the program units, which might introduce bugs and require additional comprehension effort.

6.3. Debugging and performance optimization

As an important disadvantage, Pig does not provide mature and sophisticated debugging or performance optimization techniques. Debugging Pig is mostly based on print statements. Even though Pig Latin has keywords such as *LIMIT* and *SAMPLE* to randomly select a representative sample of the data to assist in verification and debugging, the debugging of the whole pipeline is still poorly supported. To the best of our knowledge, performance optimization techniques for Pig program are not readily available either. The performance optimization is mostly based on users' experiences and trial-and-error.

7. Limitations and Threats to Validity

This section presents the threats to validity and limitations of our research.

External validity

We chose to present three software studies with Pig. Although the three experiments have a different motivation, they are all based on mining version control repositories. Prior research identifies eight major types of MSR studies [17], but our three software studies are only related to five of them (i.e., Metadata analysis, Static source code analysis, Source code differencing, Software metrics and Clone detection). Our findings may not generalize to other software studies. This threat can be countered only by performing more software studies with Pig in practice.

Construct validity

Our research can include subjective bias. For example, most program units developed in our case studies re-use modules of J-REX, which was developed by the authors. Using our own tool as a case study may cause subjectivity bias. However, in practice one will typically only alter the source code of familiar systems. In addition, the experience and performance measurements in Section 6 are based on our optimized implementation of the original, MapReduce-based and Pig-based J-REX. Even though we tried our best to optimize the implementations to gain better performance, our implementations may not be the optimal ones. We plan to further optimize our implementations and report the performance in our future work.

Moreover, program units in Pig are developed in Java. Researchers that develop ETL tools in other programming languages, such as Python, may not have the same experience as us. However, the existing tools for software studies can be re-used by wrapping them in a Java program.

8. Related Work

In this section, we discuss the related work of this paper in three areas of software engineering research.

8.1. Domain-Specific Languages for software engineering research

Various languages are currently used in software engineering research. The Grok language [21], developed by Holt, is based on binary relational algebra for the purpose of studying software architecture. Using the Grok language, software repository data can be stored in a fact database for analysis [21]. Emden *et al.* [22] used the Grok language to detect *code smells*, such as code duplication, as indicators of bad code quality and the necessity of code refactoring. A number of other researchers [21] use the Grok language to tackle software architectural and software analysis problems.

The Dependence Query Language (DQL) [23] is a Domain-Specific Language developed by Wang *et al.* to locate source code that depends on other source code, such as "component A depends on components B and C". DQL performs queries on a pre-extracted System Dependence Graph from the source code to locate subgraphs matching the query patterns, then it uses text constraints to further refine the query results.

Emden *et al.* [22] evaluated their approach using Grok on one Java program (46 KLOC), whereas DQL [23] was evaluated on four versions of two different open source projects (130 KLOC in total). However, we deal with substantially larger data such as the Eclipse source code (7,375 KLOC). Neither Grok, nor DQL has built-in distribution or multi-threading techniques. Ad hoc programming is required to scale these two languages to large-scale MSR studies. Pig, on the other hand, provides much better scalability out-of-the-box.

8.2. MSR platforms

Kenyon [24] is a data extractor for different kinds of source control systems. Kim *et al.* combined data extracted by Kenyon with CC-Finder [25], a code clone detector, and a location tracker that tracks code clones across versions [26] to perform Clone Genealogy Analysis [26].

Relational databases and SQL are widely used as platform for MSR studies. For example, FLOSSMole [27] is a public relational database that contains data extracted from a large number of software repositories. Many researchers use FLOSSMole as a platform. For example, Herraiz *et al.* [28] used data in FLOSSMole [27] to perform analysis to illustrate that most of the software projects are governed by short term goals rather than long term goals.

Alitheia Core [29], developed by Gousios *et al.*, is a platform for software quality analysis. The platform stores extracted software engineering data in a database and enables software engineering researchers to develop extensions for their customized experiments on the extracted data.

Even though software engineering researchers perform experiments on the above platforms, none of the platforms is designed for experiments with large-scale data. Researchers suffer from either the fact that experiments need a long time to finish or cannot even be performed on such platforms. Since Pig is built on Hadoop, it can scale up easily, even using commodity hardware.

8.3. Scaling software engineering research

There is some existing work on scaling software engineering research, typically involving ad hoc distributed programs. D-CCfinder [7], for example, scales CC-Finder [25] to run in an ad hoc distributed environment to support large-scale clone detection. D-CCfinder reduces the running time for detecting code clones in the FreeBSD source code from 40 days to 52 hours on an 80-machine cluster. However, such an ad hoc distributed platform requires additional programming and maintenance effort that most software engineering researchers are not interested in. In contrast to ad hoc distributed platforms, MapReduce-based data preparation platforms are scalable and general-purpose [12].

In our previous work, Hadoop, a MapReduce implementation, is used to enable large-scale MSR studies [10, 11]. Even though the MapReduce platform requires much less effort in programming and maintenance than ad hoc distributed platforms, researchers still need to transform software engineering research tools into Map and Reduce steps, and such tools are hard to debug on MapReduce platforms with large-scale software engineering data. Pig provides a scripting language that is automatically compiled to Hadoop code, to improve ease of programming. Even though Java programming is still required in Pig, the programming focuses mainly on the process of the MSR study and reuse of existing modules.

Other than Hadoop and Pig, some other techniques have been proposed to provide high-level languages and techniques and are also possible candidates to scale software engineering research. FlumeJava [30] and PLINQ [31] are examples of such techniques. FlumeJava provides a Java library to support processing data in distributed computing environment in parallel. Java programs can leverage such a library to enable large-scale data analysis by parallelizing the processing. Similarly, PLINQ introduces parallel data operations into the languages of the .NET framework, but does not support distributed computing. FlumeJava's open-source version (Plume) is still in its early development stages, while Hadoop and Pig are much more mature and have already been widely used in practice.

9. Conclusion

Traditional software analysis platforms fail to perform large-scale MSR studies with ever larger and more complex data. Even though MapReduce is capable of being a general platform for MSR studies, migrating traditional non-distributed MSR tools to MapReduce requires additional design and programming effort, and does not support reuse in practice. In this paper, we evaluate Pig, a high-level data-processing programming language on top of MapReduce, to improve the re-usability and scalability of MSR studies.

We use Pig as a data preparation (i.e., Extract, Transform, and Load) language for three MSR studies and present our implementation in detail. In addition, a performance comparison between Hadoop, Pig and a traditional non-distributed programming language shows that Pig provides similar scalability as Hadoop, which features a much lower-level distributed computing paradigm. Finally, we also report the lessons we learnt while using Pig as a data preparation language for MSR studies.

The most important advantages of Pig include the optimized data reading performance, the semi-structured data, and modular design. These Pig features support MSR researchers to prepare data for MSR studies with more flexible processes and to process large-scale data with reusable modules. However, several limitations should not be ignored, such as the large amount of boiler-plate Java code (although proportionally less than Hadoop), the effort for learning how to use Pig and the lack of debugging techniques. These limitations either cause overhead when using Pig, or prevent some MSR studies, such as real-time log mining. Despite Pig's limitations, we believe that Pig can be adopted *today* by other MSR researchers. Our experience and source code [13] can assist them in using Pig to perform large-scale MSR studies.

References

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

[2] G. Robles, J. M. Gonzalez-Barahona, M. Michlmayr, J. J. Amor, Mining large software compilations over time: another perspective of software evolution, in: MSR '06: Proceedings of the 2006 international workshop on Mining software repositories, Shanghai, China, 2006, pp. 3–9.

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

[4] A. E. Hassan, The road ahead for mining software repositories, in: FoSM: Frontiers of Software Maintenance, Beijing, China, 2008, pp. 48–57.

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

[6] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I. H. Witten, The WEKA data mining software: an update, SIGKDD Explor. Newsl. 11 (1) (2009) 10–18.

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

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

[9] C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins, Pig latin: a not-so-foreign language for data processing, in: SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, ACM, Vancouver, Canada, 2008, pp. 1099–1110.

[10] W. Shang, Z. M. Jiang, B. Adams, A. E. Hassan, MapReduce as a General Framework to Support Research in Mining Software Repositories (MSR), in: MSR '09: Proceedings of 6th IEEE International Working Conference on Mining Software Repositories, Vancouver, Canada, 2009, pp. 21–30.

[11] W. Shang, B. Adams, A. E. Hassan, An experience report on scaling tools for mining software repositories using MapReduce, in: ASE '10: Proceedings of the IEEE/ACM international conference on Automated software engineering, Antwerp, Belgium, 2010, pp. 275–284.

[12] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, in: OSDI '04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation, San Francisco, CA, USA, 2004, pp. 10–10.

[13] W. Shang, Enabling Large-Scale Mining Software Repositories (MSR) Studies Using Web-Scale Platforms, Master's thesis, Queen's University (2010).

[14] G. M. K. Selim, L. Barbour, W. Shang, B. Adams, A. E. Hassan, Y. Zou, Studying the Impact of Clones on Software Defects, in: WCRE '10: Proceedings of the 17th Working Conference on Reverse Engineering, IEEE Computer Society, Beverly, MA, USA, 2010, pp. 13–21.

[15] A. E. Hassan, Mining software repositories to assist developers and support managers, Ph.D. thesis, University of Waterloo (2005).

[16] A. E. Hassan, Predicting faults using the complexity of code changes, in: ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering, Vancouver, Canada, 2009, pp. 78–88.

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

[18] D. Crockford, The application/json media type for javascript object notation (json) (2006).

[19] CVS, http://www.cvshome.org/.

[20] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, U. Srivastava, Building a high-level dataflow system on top of Map-Reduce: the Pig experience, Proc. VLDB Endow. 2 (2) (2009) 1414–1425.

[21] R. C. Holt, WCRE 1998 Most Influential Paper: Grokking Software Architecture, in: WCRE '08: Proceedings of the 2008 15th Working Conference on Reverse Engineering, Antwerp, Belgium, 2008, pp. 5–14.

[22] E. Van Emden, L. Moonen, Java Quality Assurance by Detecting Code Smells, in: WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering, IEEE Computer Society, Richmond, VA, USA, 2002, p. 97.

[23] X. Wang, D. Lo, J. Cheng, L. Zhang, H. Mei, J. X. Yu, Matching dependence-related queries in the system dependence graph, in: ASE '10: Proceedings of the IEEE/ACM international conference on Automated software engineering, Antwerp, Belgium, 2010, pp. 457–466.

[24] J. Bevan, E. J. Whitehead, Jr., S. Kim, M. Godfrey, Facilitating software

evolution research with kenyon, in: ESEC/FSE '05: Proceedings of the 10th European Software Engineering Conference, Lisbon, Portugal, 2005.

[25] T. Kamiya, S. Kusumoto, K. Inoue, CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code, IEEE Transactions on Software Engineering 28 (7) (2002) 654–670.

[26] M. Kim, V. Sazawal, D. Notkin, G. 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, Lisbon, Portugal, 2005, pp. 187–196.

[27] J. Howison, M. Conklin, K. Crowston, FLOSSmole: A collaborative repository for FLOSS research data and analyses, International Journal of Information Technology and Web Engineering 1 (3) (2006) 17–26.

[28] I. Herraiz, J. M. Gonzalez-Barahona, G. Robles, Determinism and evolution, in: MSR '08: Proceedings of the 2008 international working conference on Mining software repositories, Leipzig, Germany, 2008, pp. 1–10.

[29] G. Gousios, D. Spinellis, A platform for software engineering research, in: MSR '09: Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories, Vancouver, Canada, 2009, pp. 31–40.

[30] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, N. Weizenbaum, Flumejava: easy, efficient data-parallel pipelines, SIGPLAN Not. 45 (2010) 363–375.

[31] M. Isard, Y. Yu, Distributed data-parallel computing using a high-level programming language, in: SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data, ACM, New York, NY, USA, 2009, pp. 987–994.