MapReduce as a General Framework to Support Research in Mining Software Repositories (MSR)

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

Abstract

Researchers continue to demonstrate the benefits of Mining Software Repositories (MSR) for supporting software development and research activities. However, as the mining process is time and resource intensive, they often create their own distributed platforms and use various optimizations to speed up and scale up their analysis. These platforms are project-specific, hard to reuse, and offer minimal debugging and deployment support. In this paper, we propose the use of MapReduce, a distributed computing platform, to support research in MSR. As a proof-of-concept, we migrate J-REX, an optimized evolutionary code extractor, 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 running time of the migrated J-REX is only 30% to 50% of the original J-REX's. This paper documents our experience with the migration, and highlights the benefits and challenges of the MapReduce framework in the MSR community.

1 Introduction

The Mining Software Repositories (MSR) field analyzes and cross-links the rich data available in software repositories to uncover interesting and actionable information about software systems [15]. Examples of software repositories include source control repositories, bug repositories, archived communications, deployment logs, and code repositories. Research in the MSR field has received an increasing amount of interest.

Most MSR techniques have been demonstrated on large-scale software systems. However, the size of data available for mining continues to grow at a very high rate. For example, the size of the Linux kernel source code exhibits super-linear growth [13]. MSR researchers continue to explore deeper and more sophisticated analysis across a large number of long-lived systems. Robles et al. reported that Debian, a well-known Linux distribution, doubles in size approximately every two years [22]. Combined with the huge system size, this may pose problems for analyzing the evolution of the Debian system in the future.

The large and continuously growing software repositories and the need for deeper analysis impose challenges on the scalability of MSR techniques. Powerful computers and sophisticated software mining algorithms are needed to successfully analyze and cross-link data in a timely fashion. Prior research focuses especially on building home-grown solutions for this. The authors of D-CCFinder [18], a distributed version of the popular CCFinder [17] clone detection tool, have improved their processing time from 40 days on a single PC-based workstation (Intel Xeon 2.8GHz, 2 GB RAM) to 2 days on a distributed system consisting of 80 PCs. Their home-grown solution is reported to contain about 20 kLOC of Java code, which must be maintained and enhanced by these MSR researchers and does not directly translate to other analyzers.

Tackling the problem of processing large software repositories in a timely fashion is of paramount importance to the future of the MSR field in general, as we aim to improve the adoption rate of MSR techniques by practitioners. We envision a future where sophisticated MSR techniques are integrated into IDEs that run on commodity workstations and that provide fast and accurate results to developers and managers.

In short, one cannot require every MSR researcher to have large and expensive servers. Furthermore, home-grown solutions to optimize the mining performance require huge development and maintenance efforts. Last but not the least, the task of performance tuning turns our attention away from the real problem, which is to uncover the

interesting repository information. In many cases, MSR researchers do not have the expertise required nor the interest to improve the performance of their data mining algorithms.

Techniques are needed that hide the complexity of scaling yet provide researchers with the benefits of scale. Research shows that scaling out (distributed systems) is always better than scaling up (bigger and more powerful machines) [20]. Off-the-shelf distributed frameworks are promising technologies that can help our field.

In this paper, we explore one of these technologies, called MapReduce [7]. As a proof-of-concept, we migrate J-REX, an optimized evolutionary code extractor, to run on the Hadoop platform. Hadoop is a popular open-source implementation of MapReduce which is increasingly gaining popularity and has proved to be scalable and of production quality. Companies like Yahoo have Hadoop installations with over 5,000 machines, and Hadoop is also used by the Amazon computing clouds. With many companies involved into its development and maintenance, Hadoop is rapidly maturing. Through a case study on the source control repositories of the Eclipse, BIRT and Datatools projects, we show that the migration effort to MapReduce is minimal and that the benefits are significant. The migrated J-REX solutions are 4 times faster than the original J-REX. This paper documents our experience with the migration and highlights the benefits and challenges of adopting off-the-shelf distributed frameworks in the MSR community.

The paper is organized as follows. Section 2 imposes requirements for a general distributed framework to support MSR research. MapReduce is explained in Section 3, as well as one of its open source implementations: Hadoop. Section 4 discusses our case study in which we migrate J-REX, an evolutionary code extractor running on a single machine, to Hadoop. The repercussions of this case study and the limitations of our approach are discussed in Section 5. Section 6 presents some related works. Finally, Section 7 concludes the paper and presents future work.

2 Requirements for a General Framework to Support MSR Research

We seek four common requirements for large distributed platforms to support MSR research. We detail them as follows:

1. Adaptability: The platform should take MSR researchers minimal effort to migrate from their prototype solutions, which are developed on a single machine.

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 paper presents and evaluates MapReduce as a possible distributed platform which satisfies these four requirements.

3 MapReduce

MapReduce [7] is a distributed framework for processing vast data sets. It was originally proposed and used by Google engineers to process the large amount of data they must analyze on a daily basis.

The input data for MapReduce consists of a list of key/value pairs. Mappers accept the incoming pairs, and map them into intermediate key/value pairs. Each group of intermediate data with the same key is then passed to a specific set of reducers, each of which performs computations on the data and reduce it to one single key/values pair. The sorted output of the reducers is the final result of the MapReduce process. In this paper, we simplify the discussion of MapReduce by assuming that mappers accept values instead of key/value pairs.

To illustrate MapReduce, we consider an example MapReduce process which counts the frequency of word lengths in a book. The example process is shown in Figure 1. Mappers accept every single word from the book, and make keys for them. Because we want to count the frequency of all words with different length, a typical approach would be to use the length of the word as key. So, for the word “hello”, a mapper will generate a key/value pair of “5/hello”. Afterwards, the key/value pairs with the same key are grouped and sent to reducers. A reducer, which receives a list of values with the same key, can simply count the size of this list, and keep the key in its output. If a reducer receives a list with key “5”, for example, it will count the size of the list of all the words that have as length “5”. If the size is “n”, it outputs an output pair “5/n” which means there are “n” words with length “5” in the book.

The power and challenge of the MapReduce model resides in its ability to support different mapping and reducing strategies. For example, an alternative mapper implementation could map each input value (i.e., word) based on its first letter and its length. Then, the reducers would process those words starting with one or a small number of different letters (keys), and perform the counting. This MapReduce strategy permits an increasing number of Reducers that can work in parallel on the problem. However the final output needs additional post-processing in comparison to the first strategy. In short, both strategies can solve the problem, but each strategy has different performance and implementation benefits and challenges.

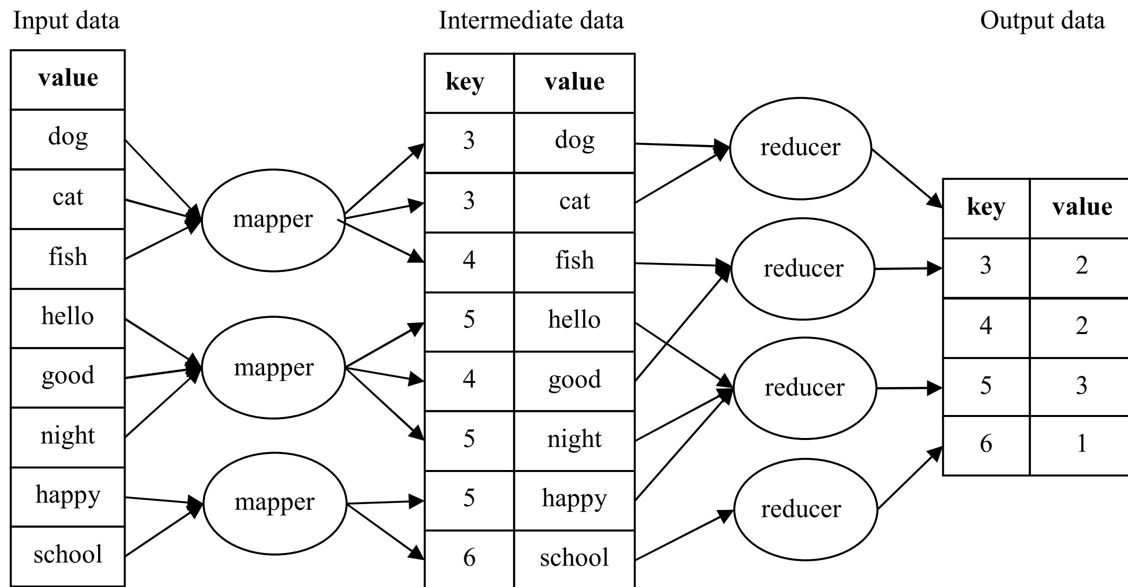

Figure 1. Example MapReduce process for counting the frequency of word lengths in a book.

Hadoop is an open-source implementation of MapReduce [3] which is supported by Yahoo and is used by Amazon, AOL, Baidu and a number of other companies for their distributed solutions. Hadoop can run on various operating systems such as Linux, Windows, FreeBSD, Mac OSX and OpenSolaris. It not only implements the MapReduce model, but also provides a distributed file system, called the Hadoop Distributed File System (HDFS). Hadoop supplies Java interfaces to simplify the MapReduce model and to control the HDFS programmatically. Another advantage for users is that Hadoop by default comes with some basic and widely used mapping and reducing methods, for example to split files into lines, or to split a directory into files. With these methods, users occasionally do not have to write new code to use MapReduce.

We used Hadoop as our MapReduce implementation for the following four reasons:

1. Hadoop is easy to use. Researchers do not want to spend considerable time on modifying their mining program to make it distributed. The simple MapReduce Java interface simplifies the process of implementing the mappers and reducers.

2. Hadoop runs on different operating systems. Academic research labs tend to have a heterogeneous network of machines with different hardware configurations and varying operating systems. Hadoop can run on most current operating systems and hence to exploit as much of the available computing power as possible.

3. Hadoop runs on commodity machines. The largest computation resources in research labs and software development companies are desktop computers and laptops. This characteristic of Hadoop permits these computers to join and leave the computing cluster in a dynamic and transparent fashion without user intervention.

4. Hadoop is mature and an open source system. Hadoop has been successfully used in many commercial projects. It is actively developed with new features and enhancements continuously being added. Since Hadoop is free to download and redistribute, it can be installed on multiple machines without worrying about costs and per seat licensing.

Based on these points, we consider Hadoop as the most suitable MapReduce implementation for our research. The next section evaluates the ability of Hadoop, and hence MapReduce, to satisfy the four requirements of Section 2.

4 Case study

To validate the promise of MapReduce for MSR research, we discuss our experience migrating an evolutionary code extractor called J-REX to Hadoop. J-REX is a highly optimized evolutionary code extractor for Java systems, similar to C-REX [14].

As shown in Figure 2, 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 for each file snapshot the Eclipse JDT parser to parse the Java code into its abstract syntax tree [1], 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,

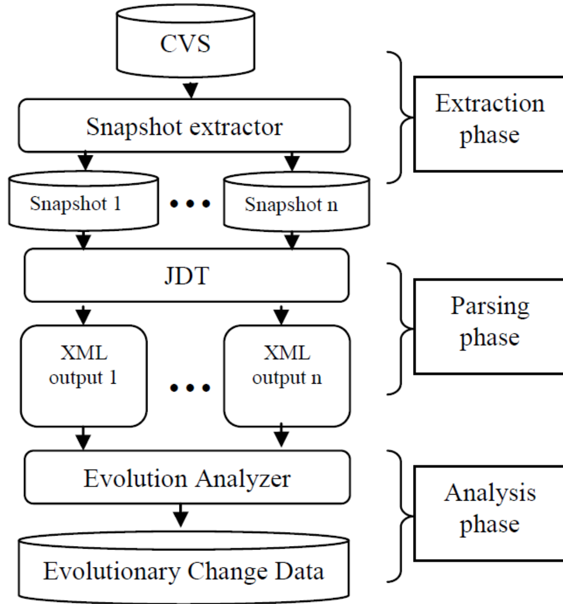

Figure 2. The Architecture of J-REX.

and generates evolutionary change data in an XML format [16]. 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"). The architecture of J-REX is comparable to the architecture of other MSR tools.

The J-REX runtime process requires a huge amount of I/O operations which are performance bottlenecks, and a large amount of computing power when 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 computation model. Through this case study, we seek to verify whether the Hadoop solution satisfies the four requirements listed in Section 2.

1. Adaptability: We explain the process to migrate the basic J-REX, a non-distributed MSR tool, to three different distributed Hadoop solutions (DJ-REX1, DJ-REX2, and DJ-REX3).

2. Efficiency: For all three Hadoop solutions, we compare the performance of the mining process among desktop and server machines.

3. Scalability: We examine the scalability of the Hadoop solutions on three data repositories with varying sizes. We also examine the scalability of the Hadoop solutions running on a varying number of machines.

4. Flexibility: Finally, we study the flexibility of the Hadoop platform by deploying Hadoop on virtual machines in a multicore environment.

In the rest of this section, we first explain our experimental environment and the details of our experiments. Then, we discuss whether or not using Hadoop for software mining satisfies the 4 requirements of Section 2.

Table 1. Disk performance of desktop and server computers.

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

Table 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	$4.2GB$	$56,851$	8 years	82,682

4.1 Experimental environment

Our Hadoop installation is deployed on 4 computers in a local gigabit network. The 4 computers consist of 2 desktop computers, each having an Intel Quad Core Q6600 @ 2.40 GHz CPU with 2 GB RAM memory, and of 2 server computers, one having an Intel Core i7 920 @ 2.67 GHz CPU with 8 Cores (Hyperthreading) 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 8 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 1. The server's I/O speed with SSD drive is twice as fast as the machines with regular disk for random I/O and turns out to be two and a half times for cached operations.

The source control repositories used in our experiments consist of the whole Eclipse repository and 2 sub-projects from Eclipse called 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 small repository with a short history. Using these 3 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 3 projects is shown in Table 2.

Table 3. Experimental results for DJ-REX in Hadoop.

Repsitory	Desktop	Server	Strategy	2 nodes	3 nodes	4 nodes
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*	—	3:02:47	—
Eclipse	—	12:35:34	DJ-REX3	—	—	3:49:05

4.2 Experiments

We conduct the following experiments:

1. Run J-REX without Hadoop on the BIRT, Datatools and Eclipse repositories.
2. Run DJ-REX1, DJ-REX2 and DJ-REX3 on BIRT with 2, 3 and 4 machines.
3. Run DJ-REX3 on Datatools with 2, 3 and 4 machines.
4. Run DJ-REX3 on Eclipse with 4 machines.
5. Run DJ-REX3 on BIRT with 3 virtual machines.

Only DJ-REX3 is applied in the last three experiments, because the experimental results for the smallest system, i.e. BIRT, already showed a significant speed improvement compared to the other two distributed strategies and the original, undistributed J-REX. The results of all five experiments are summarized in Table 3 and are discussed in the next section. The row with DJ-REX3* corresponds to the experiment that has DJ-REX3 running on 3 virtual machines.

4.3 Case study discussion

This section uses the experiment data results of Table 3 to discuss whether or not the various DJ-REX solutions meet the 4 requirements outlined in Section 2.

Adaptability

Table 4 shows the implementation and deployment effort required for DJ-REX. We first discuss the effort devoted to porting J-REX to Hadoop. Then we present the experience about configuring Hadoop to add in more computing power. The implementation effort of the three DJ-REX solutions decreases as we got more acquainted with the technology.

Table 4. 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

Table 5. 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

Easy to experiment with various distributed solutions

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 re-implementing it), then the chances of adopting Hadoop 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 distributed.

We found that applications are very easy to port to Hadoop. First of all, 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. Second, 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. Third, several code examples are available to show users how to write MapReduce code with Hadoop [3].

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. The programming logic of J-REX itself barely

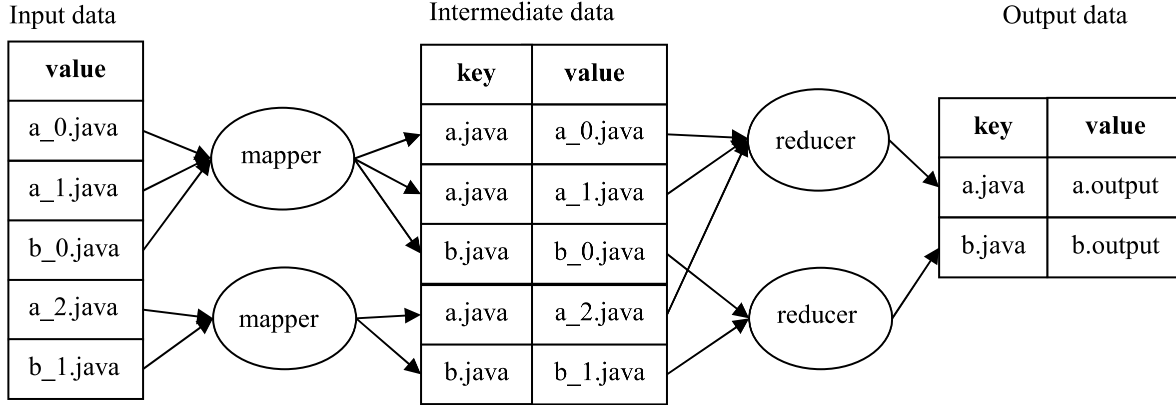

Figure 3. MapReduce strategy for DJ-REX.

changed.

The remainder of this section explains our three DJ-REX MapReduce strategies. Intuitively, we need to compare the difference 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 difference 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, which generates extra overhead for the mining process, and turned out to make the process much longer. The failure 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. 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 file names and revision numbers 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 3 flavors of DJ-REX (Table 5). Each flavor distributes a different combination of the 3 phases of the original J-REX implementation (Figure 2). The first flavor 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 it to analyze the change information. In this case, only 1 phase of J-REX becomes distributed. For DJ-REX2, one more phase,

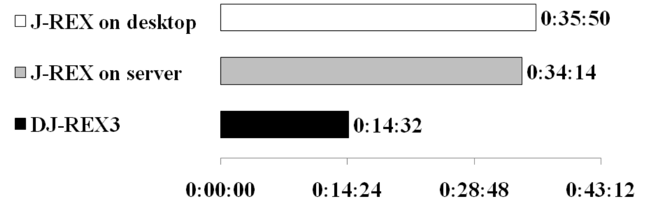

Figure 4. Running time comparison of J-REX on a server machine and desktop machine compared to the fastest DJ-REX3 for Datatools.

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 3 phases running in a distributed fashion inside each reducer. The input for DJ-REX3 is the raw CVS data and Hadoop is used throughout all 3 phases.

Using files to partition input data is an intuitive and often the most suitable option for many mining techniques which want to explore the use of Hadoop.

Easy to deploy and add more computing power

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 Hadoop on those machines. Based on our experience, we feel that porting J-REX to Hadoop is easy and straightforward, and for sure easier and less error-prone than implementing our own distributed platform.

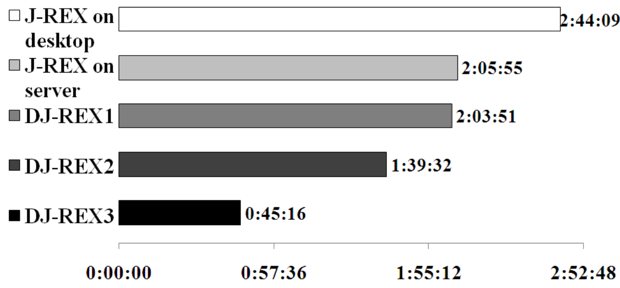

Figure 5. Running time of J-REX on a server machine and desktop machine compared to the fastest deployment of DJ-REX1, DJ-REX2 and DJ-REX3 for BIRT.

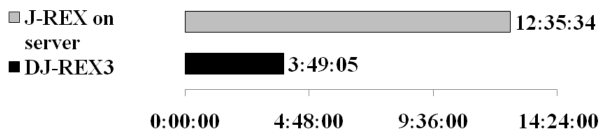

Figure 6. Running time of J-REX on a server machine compared to DJ-REX3 with 4 work nodes for Eclipse.

Efficiency

We now use our experimental data to test how much time could be saved by using Hadoop for the mining process. Figure 4 (Datatools),Figure 5 (BIRT) and Figure 6 (Eclipse) present the results of Table 3 in a graphical way.

From Figure 4 (Datatools) and Figure 5 (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 or even better than the J-REX solutions regardless of the difference in hardware machinery. As shown in Figure 5, 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 of DJ-REX is not significant. DJ-REX2 and DJ-REX3, however, outperform the server. The running time of DJ-REX3 on BIRT is almost one quarter 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 to around a quarter of the time of the server solution. It is clear that the more we distribute our process, the less time is needed.

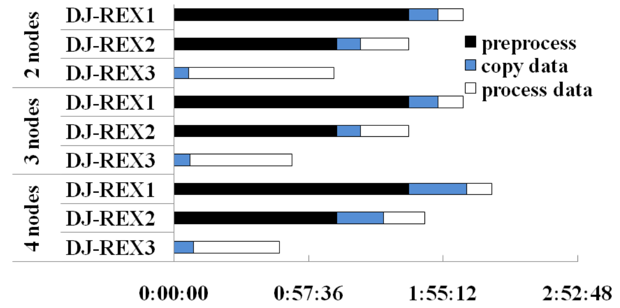

Figure 7. Comparison of the running time of the 3 flavors of DJ-REX for BIRT.

Figure 7 shows the detailed performance statistics of the three flavors 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 7, 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, whereas in DJ-REX3 everything is distributed. Hence, the fully distributed DJ-REX3 is the most efficient one.

In Figure 7, process data time (white) is decreasing constantly. The MapReduce strategy of DJ-REX is basically dividing the job by files which are processed independently from each other in different mappers. Hence, one could approximate the job's running time by dividing the total processing time by the number of Hadoop nodes. The more Hadoop nodes there are, the smaller the incremental benefit of extra nodes. In addition, a new node introduces more overhead, like network overhead or distributed file system data synchronization. Figure 7 clearly shows that copy data time (light blue) is increasing when adding nodes and hence that the performance with 4 nodes is not always the best one (e.g. for Datatools).

Our experiments show that using Hadoop to support MSR research is an efficient and viable approach that can drastically reduce the required processing time.

Scalability

Eclipse has a large repository, BIRT has a medium-sized repository and Datatools has a small repository. From Figure 4 (Datatools), Figure 5 (BIRT), Figure 6 (Eclipse) and Figure 7 (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

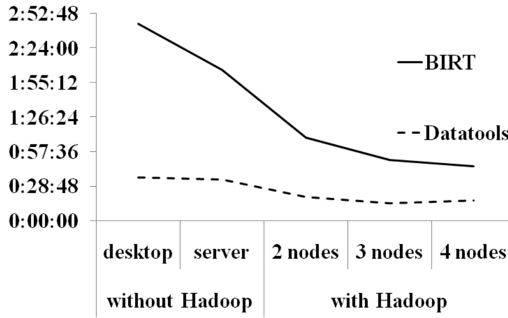

Figure 8. Running time comparison for BIRT and Datatools with DJ-REX3.

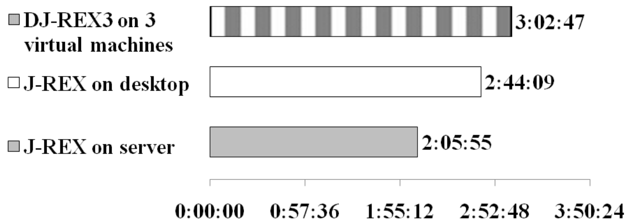

Figure 9. Running time of the basic J-REX on a desktop and server machine, and of DJ-REX-3 on 3 virtual machines on the same server machine.

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 8 shows that Hadoop scales well for different numbers of nodes (2 to 4) for BIRT and Datatools. We did not include the running time for Eclipse because of its large variance 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 6 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 amount of computing resources introduced. From Figure 8, we observe that adding a fourth node introduces additional overhead to our process, since copying input data to another node out-weighs the parallelizing tasks to more machines. The optimal number of nodes depends on the mining problem and the MapReduce strategies that are being used, as outlined in Section 3.

Flexibility

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 nodes running in 3 virtual machines on the Intel Quad Core server machine. 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 network latency. Figure 9 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. However, this could be improved by using a redundant storage array (RAID), or a networked storage array, but this is future work. 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.

5 Discussion and Limitations

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.

Incremental processing

Incremental processing is one possible way to deal with large repositories and extensive analysis. Instead of processing the data from a long history in one shot, one could incrementally process the data on a weekly or monthly basis. However, incremental processing requires more sophisticated designs of mining algorithms, and sometimes 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 low cost of migrating an analysis technique to MapReduce is negligible compared to the complexity of migrating a technique to support incremental processing.

One-time processing

One-time processing involves processing a repository once, and then storing it in a compact format for subsequent querying and analysis. Clearly, the cost of one-time processing is not a major concern. However, we believe that MapReduce can help in two ways: 1) scaling the number of possible systems that can be analyzed, and 2) speeding up the prototyping phase. Using a MapReduce implementation, analyzing and querying a large system is simply faster than when doing one-time processing without MapReduce. Moreover, although one-time processing might require a single pass through the data, it is often the case that the developers of the technique explore a lot of ideas as they are prototyping their algorithm and ideas, and have to debug the technique. The repositories must be analyzed time and time again in these cases. We believe MapReduce can help speed up the prototyping phase and offer researchers more timely feedback on their ideas.

Robustness

MapReduce and its Hadoop implementation offer a robust computation model which can deal with different kinds of failures at run-time. If certain nodes fail, the tasks belonging to the failed nodes are automatically re-assigned to other nodes. All other nodes are notified to avoid trying to read data from the failed nodes. Dean et al. [7] reported that MapReduce clusters with over 80 nodes 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.

Current Limitations

Most of the current limitations are imposed by the implementation of Hadoop. 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 nodes 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 gives much overhead. As Figure 7 and Figure 8 show, running time with 4 nodes may not be the shortest one. Finding out the optimal Hadoop configuration is future work.

6 Related Work

Automated evolutionary extractors, optimized mining solutions and distributed computing platforms are the three areas of research most related to our work.

Automated evolutionary extractors

Hassan developed an evolutionary code extractor for the C language called C-REX [14]. The Kenyon framework [6] combines various source code repositories into one to facilitate software evolution research. Draheim et al. created Bloof [8], by which users can define custom evolution metrics from CVS logs. Alonso et al. developed Minero [5] which uses database techniques to integrate and manage data from software repositories. Godfrey et al. [13] developed evolutionary extractors that use metrics at the system and subsystem level to monitor the evolution for each release of Linux. In addition, Qu et al. [23] developed evolutionary extractors that track the structural dependency changes at the file level for each release of the GCC compiler. Gall et al. [10, 11] have developed evolutionary extractors that track the co-change of files for each changelist in CVS. Gall et al. [9] developed extractors which track source code changes. Zimmermann et al. [24] present an extractor which determines the changed functions for each changelist. All these tools can be easily ported to the MapReduce framework.

Optimizing Mining Solutions

To the authors' knowledge, there is only one related work which tries to optimize software mining solutions on large scale data, i.e. D-CCFinder [18, 19]. D-CCFinder is a distributed implementation of CCFinder to analyze source code with a large size and long history in a relatively short time. Unfortunately, this implementation is homegrown and specialized to CCFinder, not open to other MSR techniques. More recently, the researchers behind D-CCFinder proposed to run CCFinder on a grid-based system [19].

Distributed Platforms

There are several distributed platforms that implement MapReduce. The prototypical one is from Google [7]. The Google platform makes use of Google's file system, called GFS [12]. Phoenix is another implementation of the MapReduce model [21]. Phoenix's main focus is on exploiting multi-core and multi-processor systems such as the Playstation Cell architecture. GridGain [2] is an open source implementation of MapReduce, but its main disadvantage is that it can only process data which can be stored in a JVM heap space. For the large size of data that we usu-

ally process in MSR, this is not a good choice. We chose Hadoop due to its simple design and its wide user base.

7 Conclusions and Future Work

A scalable software mining solution should be adaptable, efficient, scalable and flexible. In this paper, we propose to use MapReduce as a general framework to support research in MSR. To validate our approach, we presented our experience of porting 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 software mining solutions. Our experiments show that running our optimized solution (DJ-REX3) on a small local area network with 4 nodes requires 75% less time than the time needed when running it on a desktop machine and 66% less time than on a server machine.

One of our future goals is to dynamically control the computation resources in our lab. If someone wants to pull a machine out of the platform, we don't want to reconfigure the whole platform. If the machine becomes idle, one should be able to plug it back into the platform. In addition, we also plan to experiment with other technologies like HBase [4] to improve the current Hadoop deployment.

References

[1] Eclipse jdt. http://www.eclipse.org/jdt.

[2] Gridgain. http://www.gridgain.com.

[3] Hadoop. http://hadoop.apache.org.

[4] Hbase. http://hadoop.apache.org/hbase/.

[5] O. Alonso, P. T. Devanbu, and M. Gertz. Database techniques for the analysis and exploration of software repositories. In *Proc. of the 1st Workshop on Mining Software Repositories (MSR)*, 2004.

[6] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey. Facilitating software evolution research with kenyon. In *Proc. of the 10th European Software Engineering Conference (ESEC/FSE)*, 2005.

[7] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1), 2008.

[8] D. Draheim and L. Pekacki. Process-centric analytical processing of version control data. In *Proc. of the 6th Int. Workshop on Principles of Software Evolution (IWPSE)*, 2003.

[9] B. Fluri, H. C. Gall, and M. Pinzger. Fine-grained analysis of change couplings. In *Proc. of the 4th Int. Workshop on Source Code Analysis and Manipulation (SCAM)*, 2005.

[10] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proc. of the 14th Int. Conference on Software Maintenance (ICSM)*, 1998.

[11] H. Gall, M. Jazayeri, and J. Krajewski. Cvs release history data for detecting logical couplings. In *Proc. of the 6th Int. Workshop on Principles of Software Evolution (IWPSE)*, 2003.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proc. of the 19th ACM symposium on Operating Systems Principles (SOSP)*, 2003.

[13] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proc. of the 16th Int. Conference on Software Maintenance (ICSM)*, 2000.

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

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

[16] A. E. Hassan and R. C. Holt. Using development history sticky notes to understand software architecture. In *Proc. of the 12th IEEE Int. Workshop on Program Comprehension (IWPC)*, 2004.

[17] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7), 2002.

[18] S. Livieri, Y. Higo, M. Matushita, and K. Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder. In *Proc. of the 29th Int. conference on Software Engineering (ICSE)*, 2007.

[19] Y. Manabe, Y. Higo, and K. Inoue. Toward ecient code clone detection on grid environment. In *Proc. of the 1st Workshop on Accountability and Traceability in Global Software Engineering (ATGSE)*, December 2007.

[20] A. Michael, F. Armando, G. Rean, D. J. Anthony, K. Randy, K. Andy, L. Gunho, P. David, R. Ariel, S. Ion, and Z. Matei. Above the clouds: A berkeley view of cloud computing. Technical report, University of California, Berkeley, 2008.

[21] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proc. of the 13th Int. Symposium on High Performance Computer Architecture (HPCA)*, 2007.

[22] G. Robles, J. M. Gonzalez-Barahona, M. Michlmayr, and J. J. Amor. Mining large software compilations over time: another perspective of software evolution. In *Proc. of the 3rd Int. workshop on Mining software repositories (MSR)*, 2006.

[23] Q. Tu and M. W. Godfrey. An integrated approach for studying architectural evolution. In *Proc. of the 10th Int. Workshop on Program Comprehension (IWPC)*, 2002.

[24] T. Zimmermann, S. Diehl, and A. Zeller. How history justifies system architecture (or not). In *Proc. of the 6th Int. Workshop on Principles of Software Evolution (IWPSE)*, 2003.