An Empirical Study On Leveraging Logs During Bug Fixes

Suhas Kabinna, Weiyi Shang, Ahmed E. Hassan
School of Computing
Queen's University
Kingston, Ontario, Canada K7L 2N8
{kabinna, swy, ahmed}@queensu.ca

ABSTRACT

Logging is a practice used by software developers to record and convey information during the execution of a system. Logs can be used to output the behavior of the system when running, to monitor the choke points of a system, and to help in debugging the system.

Though much research has been done on the analysis of logs, most of the studies looked at either characterizing logs based on their usage or how to improve logs so theu are more meaningful. But, there has been no study so far which looks into how effective logs are in the software development process and especially in debugging.

To answer this question in our paper we first try to find co-relation between log updates and bug fixes. This is an intuitive step, because when developers fix bugs they update the logs associated with the bug or add new logs to help them fix bugs. We verify this claim on 3 large scale systems like Hadoop, HDFS and Qpid from the Apace Foundation. We then find the types of changes developers make to logging statements during bug fixes. The changes can be made

in the log variables or in the textual part or in the log severity levels. Finally, we look at how long it takes for bugs to get fixed, the number of developers involved in the fix and the comments associated with the fixes, all with respect to bugs without logging changes. We find that in all projects logging actually helps in the debugging process and three factors - time, developers and comments - are lesser when bugs have log churn associated with them. This demonstrates the importance logs play in bug fixes and motivates developers to log more.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous ; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Keywords

ACM proceedings, LaTeX, text tagging

1. INTRODUCTION

Logging is a software practise used by developers to record information during the execution of a system. Logging can be done through simple 'printf' statements or through the usage of logging libraries. When using logging libraries, each log contains a textual part which gives information about the context, a variable part which contains information about the events, and a logging level which helps categorize the severity of the log.

```
LOG.level("Message 1 " + Variable + "Messaged 2 ");
```

Logs are used for detecting anomalies [17, 21, 22], monitoring performance of the systems [4], maintenance of large scale systems [17] etc. With the increasing complexity of software systems it is difficult for developers to understand the functionality of every module, or to diagnose a system error. There is also a new market for log maintenance applications like Splunk [4], XpoLog [20], Logstash [9]which assist developers in analyzing logs.

Due to this increase in size and complexity, logging messages are used very extensively to help developers in the development process. For example, in JIRA issue HBASE-3403 (commit 1056484), we see that developers try fixing a bug where a module does not exit after a failure. The developer provides logs in the description and explains that he will go through logs to understand how this particular was state was obtained. From the logs they identify where the system fails, but fail to identify the conditions which lead to the bug. One of the developer quotes *' The log does not have important transition points for the above cited problematic region nor for the few that are stuck PENDING_CLOSE on end of the log. Let me try reproduce'*. After reproducing the bug in an unit tests, they manage to fix the bug. This bug being of 'Blocker' category (highest level) takes over 7 days to get fixed. We also observe that they update the old logging statements during the fix so all future bugs in the segment can be resolved quickly.

Previous research has already shown logs are used by developers extensively during the development of software systems. There has been research done where the logging practices are characterized based on the functions they fulfill [24].

The functions can be to provide information about control flow, to record events for tracing and also to debug error.

Most of the previous work was done on manual sampled analysis of systems. The test cases were done on a single project and no work has been to understand the leverage of logs during bug fixes in particular. One of the primary reasons is because, it is difficult to know when logs are being leveraged during bug fixes.

In our paper we first try to establish that logs are modified along with bugs. Previous research has shown that files which have high churn are more prone to defects [11, 12]. This means that developers also change logs also during bug fixes so that subsequent bugfixes are simpler. We explore such changes to logs through the help of the following research questions:

RQ 1: How often are logs leveraged during bug fixes compared to other development tasks?

We found that logs are modified more frequently during bug fixes. The effect size [7] was in the small range for log modifications in Hadoop and Qpid. We found that even log addition was in small range for Qpid. This shows that during bug fixing operations, logs are modified or added as they are closely related to the bug. It also shows that, developers have fair idea which modules can lead to bugs so they have log statements to assist them and later update them accordingly.

RQ 2: What types of modifications are more frequent to logs during bug fix?

We identified four types of log modifications from our case studies. They are 'Logging level change', 'Text Modification', 'Variable Addition' and 'Restructuring'. We found that all types of modifications, excluding Logging level change are statistically significant and have medium to high effect sizes. This shows that developers believe that adding more information about context helps in resolving issues faster.

RQ 3:Are logs useful in bug fixes?

We found logs help in quick resolution of bugs with less developer involvement. We found that bug fixing commits with log changes, have higher total churn. This means that logs are changed frequently in complex bugs. We used total churn as a controlling measure and found that given two bugs of similar complexity, the one with log changes takes less time to get resolved, less developer involvement and less discussions posts on JIRA.

We then performed a manual study where we sampled over 255 bug fixing commits with log churn to understand the usefulness of logs. We found that over 60% of commits directly used logs to assist in the debugging process. Over 51 % of those used the textual and variable objects present in logs to help in the debugging process. We also found that developers use both textual and variable part to solve more complex bugs. This study proved that logs are useful in the debugging process and developers tend to use logs to find the condition and rationale of bugs to help them in debugging.

Our study highlights that logs are useful in the debugging process and developers need to dedicate more time towards logging as and follow good logging practices.

The rest of this paper is divided as follows. Section 2 is the related work in field of logging, Section 3 contains details of our methodology in gathering and extracting the data for our study. We also present an overview of the statistical techniques we will be using in this paper. Section 4 contains the details case studies and the results we obtained. Section 5 will contain the limitations of our study and any threats to validity. Section 6 will contain further work we intend to do and will conclude the paper.

2. RELATED WORK

In this section we look at (1) the different usage of logs in improving software quality, (2) the tools which help in logging (3) the manual sampled studies done so far on analysis and characterization of logs, so we can compare the results from our empirical study.

2.1 Logs in software development process

The purpose of this section is to highlight the research that shows logs are used in debugging process. This is related to our work because (1) we show that logs are used in all the processes of a software life cycle, (2) how logs are useful in the improving the quality of the software.

Work has been done on mining execution logs in Big Data Analytic (BDA) applications to find difference in behavior of the system, when tests are run on small testing data and actual real world data [15]. Similar work has been done to show how mining execution logs in large systems can help in detecting anomalies and ensure good load tests [10]. In the paper execution logs are mined and event pair are formed. By finding patterns in these event pairs dominant behavior of the system is identified. Any event which does not follow this is tagged as anomaly and can be reported to the developers. These papers show that logs are not used only for the purpose of debugging but can assist a developer in a variety of tasks.

Research has been to detect anomalies by invariant mining [23]. This paper focuses on constructing structured logs from console logs. After construction and grouping of log messages, invariants are mined. Any log which violates a invariant is considered to be an anomaly. Unstructured log analysis has shown to be useful in detecting in even Distributed Systems Several [19]. In the paper a finite state automaton is built and trained to automatically detect performance bugs in the system. These papers consider print messages and other unstructured logs to extract information. Then they cluster or categorize the logging messages into groups and logs which fall into this cluster are treated as anomalies. Tools have also been developed to analyze system logs to obtain state-machine view, control and data flow models and other related statistics. SALSA [18] helps in deriving failure diagnosis techniques and visualization for different workloads in Hadoop project by mining execution logs. In our study, though we use the same system software (Hadoop),we try to understand how logs can be useful in debugging and not on trying to find the bug itself.

2.2 Improvement of logging

In this section we try and find out how developers log. Is it done manually or do developers use established logging libraries. From our work on studying logging libraries in the Apache foundation we know that Slf4j, Log4j and Jakarta common logging are the most used libraries for logging. This helps us to find the logging statements in our case studies. Additionally there are many tools which help developers in logging [**?**]. Log Enhancer helps to enhance existing logs by adding more values to give more understanding to the developers. This was tested with earlier version of the software and it was observed that 95% of logs added by the tool

Table 1: Overview of the Data

Projects	Hadoop		Hbase		Qpid	
	Buggy	Non-Buggy	Buggy	Non-Buggy	Buggy	Non-Buggy
# of Rev	7366	12300	5149	7784	1824	5684
Code Churn	409K	3.2M	1.4M	2.18M	175k	2.3M
Log Churn	4,311	23,838	4,566	12,005	597	10,238

were added by developer's overtime. This paper highlighted that do not log all the details in the first deployment and modify the logs in later revisions. This conforms to our findings as well, as we found that developers modify logs during debugging than other software process.

2.3 Empirical studies on logs

We finally look at other research work where logging practices have been analyzed to find where logs have been useful. Logs have been characterized in open-source systems and it has been shown that developers take multiple attempts to write correct log messages [24]. In this this paper authors found that logs are actively maintained, they are changed more often and can help in diagnosing production-run failures. Our paper explores similar aspects, but we try to understand how logs are useful in debugging process in particular.

Similar to the previous paper research has been done to study the logging practices in Industry [6]. This paper identified the most common places developers places logs in Industry. They found 5 places where developers use logs and found the reasoning behind these decisions. The results were evaluated by professionals from the industry and F-score of over 95% was achieved in predicting where logs are written.

There has been substantial work in also trying to bridge the gap between operators of the software system and developers [13]. This is crucial as logs act as mediators in this regard, as they help in providing the operational information for the mediators and for developers it helps them track the system flow.

Similar to previous work, research has also been done to understand the needs of operators and administrators [8]. In the work, by manually analyzing all email threads in three large open-source projects, the authors find 5 different types of development knowledge that are sought by practitioners from logging statements.

There has been study also done on Hadoop and Darkstar file systems where console logs are mined for identifying performance and system bugs [22]. This paper looked into two large scale systems and parsed over 24 million lines of logging data. It utilized both source code parsing and console parsing to construct meaningful log messages. Then using Principal Component Analysis (PCA) they detect the anomalies and provide visualizations. This is similar to our work because, we also use source code parsing in finding the logs. But instead of trying to build structured logs we try to classify them using levenshtein distances.

Other studies have looked at the how logs change in large scale systems and how the information conveyed by these logs change over time [14]. In this paper the author explores how communicated information through through logs are changed often and 40%-60% can be avoided. It also shows, that CI with implementation details are changed more often.

We also know logging practices can also assess the code quality of platform software [16]. This paper showed that logging characteristics can help in predicting defect prone modules in software, as developers tend to log more when they feel a module has a higher chance to be prone to bugs. This paper looked at Hadoop and Jboss and found log related metrics complement traditional metrics and increase the explanatory power of defect proneness by 40%. The other imortant aspect of this paper is the manual analysis done to identify the different reasons being logging statements. We did similar manual analysis in our paper to find the types of logging changes that occur (section 4.2) and to understand the different usage of logs in debugging (section 5).

All this prior work tries to look at usage of logs from different perspectives but none of them attempt to measure if logs are actually helpful in the development process, especially in debugging. Some of the work is done through manual analysis and has not been done on different systems to verify the results.

3. METHODOLOGY

In this section, we describe our method for data-gathering to answer our research questions and some of our assumptions when analyzing the data.

3.1 Case Study Setup

The aim of this paper is to understand what roles logging statements play in bug fixes. To answer this question, we conduct a case study on three projects in Apache Software Foundation. We found that Hadoop, Hbase and Qpid have extensive logging and best suit our case study.Table 1 highlights some of the statistics of these systems.

Hadoop (http://hadoop.apache.org/)

Hadoop is an open source software framework for distributed storage and distributed processing of Big data on clusters. It uses the MapReduce data-processing paradigm. We used Hadoop releases 0.16.0 to 2.0.

HBase (http://hbase.apache.org/)

Apache HBase is a distributed, scalable, big data store using Hadoop database. We used Hbase releases 0.10 till 0.98.2.RC0. This covers more than 4 years of development in Hbase from 2010 till 2014.

Qpid (https://qpid.apache.org/)

Qpid is an open source messaging system which implements Advanced Message Queuing Protocol. Qpid we used all data from 0.10 till 0.30 release. This covers development from 2011 till 2014.

We used Apache software versioning system, SVN to collect the commit history for all the projects in our case study.

We used JIRA to collect the buggy reports for all the projects in our case study. We extracted the reports in XML format so its easy to parse in our scripts.

3.2 Data Collection and filtering

Using SVN diff we extract the commit history for all the projects in our case study. From this extracted data we calculate the metrics necessary to answer our research questions. As we are trying to find the logging changes that occur in each commit, identifying logging messages is the most essential part of our study which is explained below.

3.2.1 Log Identification

To identify the logging changes in the data, we manually looked at some of the commits to find common pattern in logging. After identifying these patterns we used pattern based searching through our scripts, to identify the log messages automatically. Some patterns were specific to each project and some uniform across projects. This uniformity is due to the use of the same logging libraries across projects.

Project Specific pattern:
QPID_LOG(error, "Rdma: Cannot accept new connection (Rdma exception): " + e.what());
Logging Library pattern:
log.debug("public AsymptoticTestCase(String " + name + ") called");

After identifying the logs, the next important step is matching the buggy JIRA issues to its corresponding commit. In SVN, all the commits tagged to their corresponding JIRA issue [1]. We extracted all the commit (revision) numbers and the JIRA issues for the commit, and matched it against the list of Buggy JIRA issues. From this we obtained list of Buggy and Non-Buggy commits.

3.2.2 Data filtering

After obtaining the 2 data sets we had to filter out the noise to achieve good results. The first filtering activity was to calculate churn for 'JAVA' class files (non Java files act as noise), in all the projects. To achieve this we wrote scripts which checks if a particular commit file has '.java' files and calcualtes churn for those files.

After getting list of buggy and non-buggy commits we calculated the churn for each commit. We obtain the churn for only Java class files as all the the projects are JAVA based.

When we calculated the churn for each commit we observed some commits had very high code churn. This was because when there is branch or merge operation, most of the files are moved from one directory to another. Because, SVN diff does not track if a file is copied from different directory, the churn is very high. For example revision 952,410 in HBase project, is a branching operation where a new trunk revision 0.20.5 is created. This commit has added code churn of 119,504 and no deleted code churn.

To filter out such noisy data, we consider only those revisions which have added churn and deleted churn more than 5% of total churn. In the commit above the deleted churn is 0 so we exclude theis revision from our study. As a second safety measure to avoid noise, we also use the change-list file to remove all the branching commits. In a change-list file all commits which are related to branch or merge operations fall under 'BRANCH_SYNC' category. All such commits are excluded from our data-set. Even after this we observed that some commits have churn values over to 30,000. Most of these commits were Improvements to the projects (revision 1,334,037) and cannot be excluded.

[1]SVN Commit 1162209 - http://svn.apache.org/viewvc?limit_changes=0&view=revision&revision=1162209

3.2.3 Identifying data sets.

The churn file had 3 basic data sets namely 'Total Churn', 'Added Logs' and 'Deleted Logs'.

Total Churn: Total churn as the name implies is the total of all the added and deleted lines of code in a commit, including log lines.

Added Log Churn: This is an aggregate of all log lines added in a commit.We calculated this from the SVN commit history.

Deleted Log Churn: Like added log churn, this metric counts the total number of log lines deleted from commits.

After obtaining this data sets for buggy and non-buggy commits, we examined the added and deleted logs in the commits. The number of added logs closely matched the deleted log count in majority of the commits so examined the logs being added and deleted. The 2 data sets were similar because developers modify existing logs rather than add new logs. As SVN diff does not provide a built in feature to track changes in a file line by line, we used the Levenshtein distance [5] and ratio to find out when logs are modified. From this categorization we obtained 3 new data sets namely:

1. Modified Logs: Using Levenshtein distance we identified logging messages which were modified from existing messages i.e. we compare the 'Added Log' vs 'Deleted Log' in each commit.

2. New Logs: This set represents all those logs which were newly added in a commit. To obtain this we subtracted the added log count from the modified log count.

3. Deleted Logs: This data set represents all those logs which were deleted in a commit and not modified. Similar to new logs we subtracted the deleted log count from modified log count to get this churn.

Using this new data set we can answer the question whether logs are leveraged during bug fixes more.

4. STUDY RESULTS

In this section we derive the necessity of each research question, the approach to solve them and the results we obtained.

RQ1: How often are logs leveraged during bug fixes compared to other development tasks?

Motivation

Logs are used extensively for debugging process. But, this is not the only reason for using logging in a system. So in our first research question we try to address this issue. Answering this questions helps us understand where developers log more.

Approach

Its intuitive that logs are generally modified when bugs are fixed. This is because various research has already proved that a module that's been modified continually has higher

Figure 1: P values and Effect Size

Projects	Hadoop		Hbase		Qpid	
Metrics	P-Values	Effect Size	P-Values	Effect Size	P-Values	Effect Size
Modified Churn Buggy Vs Non Bugy	**2.88e-4**	0.167(small)	**0.0353**	0.0886	**0.0281**	0.329(small)
New Churn Buggy Vs Non Buggy	**0.00202**	0.0078	**0.00353**	0.134	**0.0032**	0.234(small)
Deleted Churn Buggy Vs Non Buggy	0.087	-0.0455	**0.00489**	0.120	**0.00952**	0.042

chance of having bugs than modules which have not been modified [3]. So we look at bug fixes which also have log modifications with them. This is done by developers so future occurrences of similar bug can be resolved easily with the updated information in the log statements.

From the data sets obtained in previous section i.e Modified, New and Deleted logs, we calculated churn for each data set. We used 'Total Churn' of a revision to control the other metrics. The 3 new metrics are:

$Modified\,Churn = Modified\,Log/Total\,Churn$

$New\,Churn = New\,Log/Total\,Churn$

$Deleted\,Churn = [\,Deleted\,Log/Total\,Churn$

To measure the statistical significance of these metrics, in buggy and non-buggy revisions we use the MannWhitney U test (Wilcoxon rank-sum test) [2]. Wilcoxon test gives p-value as the result. A p-value of ≤ 0.05 means that the test was statistically significant and we may reject the null hypothesis (i.e., the two populations are different). By rejecting the null hypothesis, we can accept the alternative hypothesis, which tells us if one population is statistically significantly larger than the other. We use Wilcoxon test because we know our metric are very skewed and if we use standard T-test the resulting p-value will be wrong. Wilcoxon test is a non-parametric test, meaning the distribution of the population does not factor into the results.

Because Wilcoxon test only tells if the two data sets are different and not how large the difference we used Cohens.d [1]z to measure the size of difference. The value of Cohens.d can range from 0 till infinity, where higher the value means bigger the difference between the data sets.

$$Effect\,Size = \begin{cases} 0.16\ < & Trivial \\ 0.16 - 0.6 & Small \\ 0.6 - 1.4 & Medium \\ 1.4\ > & Large \end{cases}$$

Results

We observe that in both Hadoop and Qpid modified churn has effect size higher than 0.16 (this is show in figure 1). This implies that developers tend to modify logging messages more frequently during bug fixes than other developmental tasks. This is done because developers do not have to think a lot before adding logs, as its already present and they have to just tweak the logs so it gives more information. This implies that in most cases the existing logging messages do not convey all the information necessary for a bug fix.

```
+ log.debug("SO_RCVBUF : %s", socket.getReceiveBufferSize());
- log.debug("default-SO_RCVBUF : %s", socket.getReceiveBufferSize());
```

Table 2: Distribution of Log Modifications

Projects	Hadoop (%)	Hbase (%)	Qpid (%)
Restructuring	82.6	61.4	55.8
Text Modification	7.85	12.1	18
Variable Modification	7.9	8.4	12.5
Logging Level Change	3.85	5.4	13.6

We also observe that New Logs have effect sizes higher than 0.13 in Hbase and Qpid. This implies in some cases developers completely ignore logging during initial development and have to spend more time starting from scratch. This is an extra effort which can be avoided by good logging practices.

For Deleted Logs we obsereve that in all 3 projects the effect sizes are smaller than 0.13. This shows that logs are never removed by developers. In Hadoop we observe that the effect size is negative which implies logging statements are removed more from non-buggy commits than buggy commits. In Qpid and Hbase even though the effect size is positive, its too small which to draw meaning full conclusions.

Finding: Hadoop and Qpid are statistically significant and have high effect size value in small range for Modified Churn.
Implication: Developers modifying existing logs more than adding new logs. This implies that, developers have fair idea which modules can lead to bugs so they have log statements to assist them. But as that these logs do not convey all the information necessary so they are modified later by the developers.

As we observe significant effect size in two of the projects, we looked into the types of logging modifications that occur to understand what changes developers make to logs.

RQ2: What are the different types of log modifications developers make during bug fixes ?

Motivation

From the previous research question we established that logs are leveraged more during bug fixes than other cases. But the effect size was still small within the 0.1 - 0.3 range. So we tried to look at the different types of logging modifications which occur.

Approach

To find the types of modifications in logging messages, we looked at the components of a log message. We observed 3 main components namely the log level (Warn, Error, Debug, Trace or Fatal), the message part and the variables part. The changes can be made to each of these parts and we

divided the logging changed based on these categories. We used Levenshtein distance [5] and ratio's to filter the logging changes into these categories. The four categories are:

- **Restructuring:** In this only white spaces are changed or the logging statement is kept intact but moved to different place in the file. We observed that this is one of the most frequent type of change done in both buggy and non-buggy commits from table 2. To categorize this we set the Levenshtein distance less than 5 **and** ratio higher than 0.9.

- **Text Modification:** In this type, message part of log is modified, keeping the variables constant. To classify this we first checked if the Levenshtein distance is less than 5 **or** Levenshtein ratio greater than 0.7. If this condition was met, we said the log message had similarities with one of the deleted logs. Then we checked if the similar part was 'message' or 'variable' part. If there was over 0.9 similarity with the variable parts but not with the text part, we concluded it was text addition or deletion. Because this condition overlaps 'Restructuring', they were classified first and those logs were excluded from further classification.

```
- Logger.warn( " Sample Text Goes Here " + printAVariable );
+ Logger.warn( " New Sample Text Goes Here " + printAVariable);
```

- **Variable Modification:** In this type, variable part of log is modified , keeping the text constant. Its classified similar to Text Modification by changing the final condition.

```
- Logger.warn( " Sample Text Goes Here " + printAVariable );
+Logger.warn( " Sample Text Goes Here " + printAVariable + printBVariale);
```

- **Logging Level Change:** In this type only change is the log level change. To find this we checked if the levenshtein distance of the entire log message is equal to the Levenshtein distance of the log level. If this condition was met, it means only the part that is changing is the level of log.

```
- Logger.warn( " Sample Text Goes Here " + printAVariable );
+ Logger.debug(" Sample Text Goes Here " + printAVariable );
```

After finding the different categories, we performed a manual analysis to find the distribution of each of these categories. The steps we followed are as follows:

- **Step 1.** Collect all the commits with log churn from both data sets (i.e Buggy and Non-Buggy), in all projects into two data set Bug_{Total} and $Non{-}Bug_{Total}$,

- **Step 2.** Randomly sample a subset both data sets Bug_{Total} and $Non - Bug_{Total}$, to get Bug_{Subset} and $Non - Bug_{Subset}$, so they have 95% confidence level and $\pm$ 5% confidence interval.

- **Step 3.** For each commit in both sets, find which category of log modification they belong to (i.e. logging level, variable modification, text modification or restructuring)

- **Step 4.** Repeat step 3 for all the commits in the subset till distribution is obtained.

Table 2 highlights the distribution of our analysis.

After categorization we find out the churn for these categories in our data-sets. We obtained 4 metrics:

Restructuring Churn

As explained above we categorized all the logging statements into 4 types. This metric counts the total number of logs of type 'Restructuring' were present in each commit.

Text Modification Churn

This metric is the aggregate oflog changes of type 'Type Modification' are present in each commit

Variable Modification Churn

This metric is the aggregate of log changes of type 'Variable Modification' are present in each commit.

Logging Level Churn

This metric is the aggregate of log changes of type 'Logging Level Change' are present in each commit.

After finding the churn in each of these categories we tried to find how significant are these compared to the non buggy revisions. To do this we created smaller subsets for buggy and non-buggy revisions which had non-zero values for each of the metrics. As in section 4.1.2 we measure the difference between the data sets through Wilcoxon tests and Cohens.d.

Results

We found that Variable Modification has higher effect size among all types of log changes. This implies that developers modify the variable part of their logs, as this helps in providing more information about the system. This is useful because if any other bugs originate in that module, the addition data can help in easy resolution. It also shows that developers do not provide all the details the first time logs are written and update them during later fixes.

We also found that Text Modification is more in bug fixes. This implies that developers value more in providing contextual data to an existing log rather than writing new logging statements.

We found log restructuring is usually done when the same log statements are moved/copied to multiple places as part of code change. We looked at over 100 commits across all three projects to understand this behavior. We observed that in majority of the cases, developers use logging statements before try, catch blocks or do not check all the conditions. When a bug is raised the logs help in fixing the issue, but along with the fix they place the original logging statement in a try catch block and handle further exceptions properly. Example of this is 792,522 revision of Hadoop. In this revision we see that the log messages are not placed in the appropriate 'try', 'catch' blocks. We also see that in some parts of the code the exceptions are not even present and later added by developers. In this commit the same logging message is copied in multiple places and hence its falls under restructuring category.

Table 3: P-Values and Effect Size for Log Modifications

Projects	Hadoop		Hbase		Qpid	
Metrics	P-values	Effect Size	P-values	Effect Size	P-values	Effect Size
Restructuring Bug Vs Non Bug	**1.69e-11**	0.260(small)	**6.33e-03**	0.2092	**9.14e-08**	0.987(med)
TextMod Bug Vs TextMod NonBug	**7.75e-04**	0.153(small)	**2.94e-05**	0.308	**4.68e-08**	0.531(small)
VariableMod Bug Vs VariableMod NonBug	**1.94e-06**	0.447(small)	**3.51e-04**	0.614	**5.19e-05**	1.209(med)
LevelChange Bug Vs LevelChange NonBug	**0.0057**	0.412	0.153	-0.05	0.341	0.396

From table 2, we see that this form of changes a major bulk of logging changes that occur in both data sets. This shows us that developers when writing code omit safe coding practices and do not use try, catch and finally blocks in their code.

Another example of this 1,042,282 revision in which a log statement is moved from the beginning of code block to the end of block. From our manual study (section 5) we found that this is mostly done to improve the readability or increase the performace. In HBASE-4288, the logs are moved into new blocks, so they are printed only when 'Trace/Debug' option is enabled.

Finding: Variable Modification has small to medium effect sizes in all projects
Implication: Developers believe adding more variable/dynamic values to logging statements helps in the debugging process. This implies that developers take more than one attempt to get log statements correct. This shows that developers can spend more time or take use of the logging tools present to help log more in first commit so subsequent changes are not necessary.

We observe that log level changes has P-values less than 0.05 only for Hadoop project. We also see that in table 2, logging level changes are the least among all the types of logging changes. This tells that logging level changes are similar in both buggy and non-buggy commits.

From these results we have firmly established that logs are used and leveraged by developers during bug fixes. We also see that new logs are added during bug fixes than other types of changes.

4.1 RQ3: Are logs useful in bug fixes?

Motivation

In our previous research questions we found that logs are leveraged more frequently in bug fixes. Next the main question is how helpful logs are in bug fixes. We also try to understand under which circumstances logs are helpful. We try to do this by using a measure and comparing our results.

Approach

To find the usefulness of logs in bug fixes we had to collect new a data-set different from previous research questions. To do this we first collected all the buggy JIRA issues for the three systems. Then using the churn metrics from previous step we split this buggy list into two segments: (1) bug fixes with log churn (2) bug fixes without log churn.

Next, we extracted 3 metrics for each JIRA issue. The metrics are:

Time Taken

This is the time taken from the time a bug is opined till its resolved or closed. For example in JIRA if bug was reported in 01-2-15 and closed in 05-2-15 it means the time taken to fix the bug was 4 days. We used this because this is a very basic measure and tells us how long it takes for a bug to be fixed.

Comments

This metric gives the total number of comments are present in a JIRA issue. We obtained this by finding the number of comment id's present in the JIRA XML file. We used this measure as it gives information about how complex issues are. If its a simple issue, it wont be discussed and will be resolved quickly. But, if its a complex bug there will be more discussion posts and takes longer to be resolved.

Developers

Every task in JIRA is generally assigned to particular developer and there number of viewers who are interested in the issue and help in resolving it. This metric gives the number of such unique developers who commented on the JIRA issue posts and were involved in resolving it. We obtained this by finding all the unique author names in the XML file. We used this as a metric as it shows human effort needed. More number of developers involved and commenting on a issue list, signifies more human effort spent resolving the bug.

Finally we matched this JIRA metrics without our two data segments based on JIRA issue numbers. The statistics of the data sets are show in table 5. We used Wilcoxon test and Cohens.d to measure the difference between the datasets.

Findings

We found that the total churn is higher for commits with log changes. Here total churn is a complexity measure, and higher churn implies the bug is more complex. From table 4, we see that in all projects its that total churn is statistically significant and has small effect sizes. In table 5 we again see this, where in every project the average churn is several magnitudes higher for bug fixing commits with log changes. This shows that developers always tend to change logs more during complex bug fixes.

Because churn is correlated to other metrics in our data set, we used total churn as a measure to calculate the importance of each metric i.e. we take ratio of each metric against the code churn. We did this because in all projects logs are changed only during a complex bug fix. Because of this, when we measure the statistical significance of the 2

Table 4: P-Values and Effect Size for Bug fixes

Projects	Hadoop		Hbase		Qpid	
Metrics	P -values	Effect Size	P -values	Effect Size	P -values	Effect Size
Total Churn	**2.2e-16**	0.563(small)	**2.2e-16**	0.093	**3.15e-08**	0.270(small)
TimeTaken_Log Vs TimeTaken_WithoutLog	**4.26e-03**	-0.145(small)	**7.44e-14**	-0.167	0.0865	-0.119
Comments_Log Vs Comments_WithoutLog	**2.2e-16**	-0.507(small)	**5.16e-11**	-0.289	**2.34e-03**	-0.227(small)
Developers_Log Vs Developers_WithoutLog	**2.2e-16**	-0.577(small)	**2.2e-16**	-0.538	**4.73e-02**	-0.375(small)

data sets, it will be biased. To correct this, we used 'total churn' as controlling measure and the results are shown in table 4.

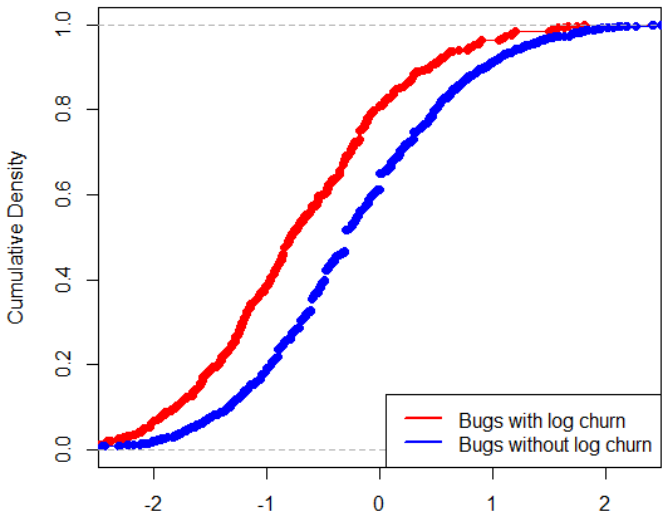

Figure 2: Cumulative Density Function of the fix time for Bug with log churn and Bugs without log Churn. The x-axis shows the time taken for resolution after its normalized using total churn. y-axis shows the cumulative density.

***Findings**: Total Churn has positive effect size **Implication**: This implies that bug fixes with log changes, have larger code churn. This means generally logging changes are generally done to more complex bugs and ignored for smaller bugs by developers.*

We found that the effect sizes are negative for all three metrics when controlled by total churn. This means that given two bugs of same complexity the one with log churn takes less time to get resolved and involves less number of developers in the fix**.** Because there is less developers involved in the discussion, there are less number of comments related to the fix. An example where logs are used is in JIRA issue HBASE-3074 and commit number 1,005,714. In the JIRA issue we see the very first comment is to provide additional details in the logging message about where the connection manager fails. When we looked at the commit, we see that the developers add new variables into an existing log message and remove another logging message as its not necessary. The developers add the name of the existing server which has gone stale, which is not previously done before. This additional data we think helps in tracing the cause of the failure of connection manager. This example further validates our findings and shows that logs are indeed very useful in the debugging process.

Table 5: Overview of the data sets with code churn

Projects	Metrics	With Log Change	Without Log Change
Hadoop	No of Commits	607	2947
	Average Churn	371	41
	Average Time	62 days	48 days
	Average comments	20	16
	Average users	7	6
HBase	No of Commits	862	2622
	Average Churn	692	295
	Average Time	36 days	26 days
	Average comments	27	19
	Average users	6	6
QPid	No of Commits	83	445
	Average Churn	1193	145
	Average Time	96 days	64 days
	Average comments	5	4
	Average users	3	2

Findings: Time Taken for resolution, Developer involvement and Number of discussion posts are statistically significant and have negative effect sizes.
Implication: This implies bugs without any log churn take longer to be fixed, need more involvement and discussions. This shows that logs play a vital role in solving bug fixes. Drawing from the manual study done in research question two, we observed that logs help in locating the buggy module and when fixes are made to the module, developers modify the logging statements. This is cyclic process and hence its vital that developers log in the initial development process.

To validate our study we did a manual study on to understand how logs are actually used in the debugging process. This is explained in further details in the subsequent section.

5. MANUAL STUDY ON BUG FIXES

From our research question we found that logs help in reducing the time and human effort necessary during the debugging process. To validate our case study, we did a manual analysis to find out where and in what scenarios logs are used. The steps are as follows

- **Step 1.** Collect all bug fixing commits with log churn from all three projects. Bug_{Total}
- **Step 2.** Randomly sample a subset of Bug_{Total} to get Bug_{Subset}, that has 95% confidence level and $\pm$ 5% confidence interval (we checked 180 samples)
- **Step 3.** For each commit in the subset find out where logs are used and in what scenarios and find patterns.
- **Step 4.** Repeat step 3 for all the commits in the subset till distribution is obtained.

Analysis Results

From this manual study we tried to answer three important questions:

1. What percent of bugs directly leverage logs in the debugging process ?

We found that 70.7% of the sample used logs directly for fixing the buggy issues. We looked into the JIRA discussion posts and we observer that over 128 of the reports, made use of log dumps, master log records or the developers in their comments mentioned logs to trace the problem. We observed that developers use logging statements to trace the bugs and even use the exceptions generated. It should be noted that in some of the JIRA discussions, developers write ' Let me trace/re-create the problem ' and in the next post they provide a list of conditions which causes the bug. In such posts we cannot know if logs are leveraged.

We observed that 34.3% of the samples (35 commits), were cases in which the logs themselves were bugs. These issues were either trivial from mis-spelt words , haphazard logs to complex issues were logs cause performance issues. Example- In 'HBASE-5081' we see that a log-splitting in function 'deleteNode' forms a race condition and hangs.

2. What type of knowledge/information do developers look in logs ?

61% of the reports (78) made use of both variable (dynamic objects) and textual part present in the logging statements to resolve the bugs. We observed that developers output the system state, the server/connection name, time, logging level, and even object names in the log lines. We found that these details are generally sought after by developers more during debugging process. Example - In HBASE-4797, developers actually provide an extract of logs generated on JIRA. We observe that developers draw many conclusions using the logs present and this helps in resolving the bug. In this particular example, developers used the time-stamp, region-servers and the sequence-id's present in the logs. We also observed that these commits were associated with higher code churn implying, developers need more information to fix more complex problems.

The remaining 39 commits used the log message themselves to diagnose the problem. We observed that majority of these bugs (48% of the commits), were caused by the logs themselves. In such cases, logs have grammatical errors (example- commit 1333321), are not placed within proper blocks (example- commit 1236977), cause performance bugs (example- commit 999046). In all these cases, the presence of log is sufficient enough information to debug the problems and developers do not need additional information.

Findings: 61% of buggy commits made use of both textual and variable part in logs and average churn was 244 LOC (lines of code) to 160 of the latter.
Implication: When there is more complex bug to be solved, developers use both the textual and variable (dynamic part) to solve the problems. This tells us that when its complex bugs, developers try to record all the information to see where fault occurs. This means that logs are useful in understanding the specific set of conditions which triggers the bug (rational behind the bug). Hence, its observed that after the fix the developers add these new variables to the commit to assist them in future.

In conclusion, this manual study we studied the leverage of logs in JIRA issue reports. We observed that 56% directly leveraged logs on JIRA (i.e. used word 'log' or provided extracts of logs). Logs are used by developers to understand the rationale behind bugs and are an asset to a developer.

6. LIMITATIONS AND THREATS TO VALIDITY

External Validity

In this study we found that logs are leveraged during bug fixes more than any other type of changes. We looked at three projects from the Apache Software Foundation namely Hadoop, HBase and Qpid. This is also limiting factor as these projects are all Java based and we haven't looked into other programming languages.

The other limitation is projects in which logging data is less. We ran experiments on several other projects like Cassandra, Cayene, Zookeeper and Lucene. In all these projects the total number of logging statements were less to draw any meaningful conclusions. When we manually examined the data for Cassandra and Lucene we observed many custom logging statements which were not caught from pattern matching. As stated above its almost impossible to catch all instances of these custom logs in projects.

Internal Validity

In our initial research questions, we tried to find if there is relation which proves logs getting changed implies a high probability of bug fix or not. This was shown to be true and we deduced that logs are in-fact changed more often during bug fixes than other types of changes. But there can be instances where logs are used in bug fix, but they are not changed as they provide sufficient information. In such cases its impossible to determine if logs were useful or not.

Although our study shows presence of logs reduces time of resolution, we do not claim any causal relationship between the two.There are many other contributing factors which need to be considered when trying to predict resolution time for bugs. The purpose of our study was to demonstrate that logs are indeed useful and motivate developers to log more during initial development.

Construct Validity

When correcting the data, we found that some of the revisions were only related to branching or merging . To eliminate this we looked at all JIRA commits which has type Branching in them and removed these branching commits from our data set. But we later found some of the commits are tagged under the category of UNKNOWN. When its under 'Unknown' category it was impossible to know if its branching or not.

The other major limitation is user defined logs in the Java Code. When searching for log lines we looked at specific patters in the files. These patters were 'Log', 'Logger', 'LOGGER', 'LOG', 'log' and few other variants. But the users can define their own names for these statements and it would make it impossible to search for such user defined functions in the change commits.

7. CONCLUSION AND FUTURE WORK

In this work we looked at Apache Hadoop, Hbase and Qpid to find out if logs are more useful in bug fixes or not.We first try to establish how we are going to measure this by checking for log modifications that are part of a bug fixing commit. In our initial research questions we find that logs are modified and added during bug fixes more than other types of developmental tasks

We identified 4 different types of logging changes namly 'Text Modification', 'Variable Modification', 'Restructuring' and 'Logging Level Changes'. We saw that 'Variable Modification had the highest effect size in all projects, followed by 'Restructuring' and 'Text Modification'. This showed that developers tend to add or delete or change variables more often during bug fixes. We also saw that developers tend to change the position of logging statements or change the text in logs more frequently during bug fixes.

Next, we studied the bug fixes with log changes and found that 1) Bug fixes with log changes take more time to get resolved, 2) Bug fixes with log changes have more comments and users in the discussion. We found that given 2 bugs of same complexity, the one without log changes takes longer time to be fixed, needs more number of developers to be involved in the fix and has long discussions post also.

Finally we did a manual analysis were we found developers use logging statements to understand the rationale behind logs. We found over 60 % of the cases directly used logs in the debugging process. We found that over 56 % of the time logs are used to understand the rationale behind bugs and find which specific conditions triggers bugs.

This proves that logs are in-fact helpful in bug fixes and are useful for quick resolution of bugs with less involvement from developers. It shows that if logs added during bug fixes the resolution time and development can reduce during subsequent bug fixes and help other developers.

8. REFERENCES

[1] Cohens d - http://www.statpt.com/appliedgen/cohend.pdf.

[2] Wilcoxon test - http://www.ime.unicamp.br/ dias/ch10.wilcoxon.pdf.

[3] *T. M. Khoshgoftaar, E. B. Allen, N. Goel, A. Nandi, and J. McMullan. Detection of software modules with high debug code churn in a very large legacy system. In Proceedings of ISSRE*, 1996, page 364. IEEE CS Press, 1996.

[4] L. Bitincka, A. Ganapathi, S. Sorkin, and S. Zhang. Optimizing data analysis with a semi-structured time series database. In *Proceedings of the 2010 Workshop on Managing Systems via Log Analysis and Machine Learning Techniques*, SLAML'10, pages 7–7, Berkeley, CA, USA, 2010. USENIX Association.

[5] L. definition : http://xlinux.nist.gov/dads/HTML/Levenshtein.html.

[6] Q. Fu, J. Zhu, W. Hu, J.-G. L. R. Ding, Q. Lin, D. Zhang, and T. Xie. Where do developers log? an empirical study on logging practices in industry. In *ICSE Companion 2014 Companion Proceedings of the 36th International Conference on Software Engineering*, pages Pages 24–33. ACM New York, NY, USA l'2014 ISBN: 978-1-4503-2768-8 doi>10.1145/2591062.2591175, 2014-05-31 2014.

[7] V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. K. Sjøberg. A systematic review of effect size in software engineering experiments. *Information and Software Technology*, 49(11-12):1073–1086, Nov 2007.

[8] U. L. L. U. D. Knowledge. Weiyi shang, meiyappan nagappan, ahmed e. hassan, zhen ming jiang. In *The 30th International Conference on Software Maintenance and Evolution (ICSME 2014)*, 2014.

[9] logstash. http://logstash.net.

[10] H. Malik, H. Hemmati, and A. Hassan. Automatic detection of performance deviations in the load testing of large scale systems. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 1012–1021, May 2013.

[11] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 284–292, New York, NY, USA, 2005. ACM.

[12] N. Nagappan and T. Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, ESEM '07, pages 364–373, Washington, DC, USA, 2007. IEEE Computer Society.

[13] W. Shang. Bridging the divide between software developers and operators using logs. In *Software Engineering (ICSE), 2012 34th International*

Conference on.

[14] W. Shang, Z. M. Jiang, B. Adams, A. E. Hassan, M. W. Godfrey, M. Nasser, and P. Flora. An exploratory study of the evolution of communicated information about the execution of large software systems. *Journal of Software: Evolution and Process*, 26(1):3–26, 2014.

[15] W. Shang, Z. M. Jiang, H. Hemmati, B. Adams, A. E. Hassan, and P. Martin. Assisting developers of big data analytics applications when deploying on hadoop clouds. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 402–411, Piscataway, NJ, USA, 2013. IEEE Press.

[16] W. Shang, M. Nagappan, and A. Hassan. Studying the relationship between logging characteristics and the code quality of platform software. *Empirical Software Engineering*, 20(1):1–27, 2015.

[17] M. Syer, Z. M. Jiang, M. Nagappan, A. Hassan, M. Nasser, and P. Flora. Leveraging performance counters and execution logs to diagnose memory-related performance issues. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 110–119, Sept 2013.

[18] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan. Salsa: Analyzing logs as state machines. In *Proceedings of the First USENIX Conference on Analysis of System Logs*, WASL'08, pages 6–6, Berkeley, CA, USA, 2008. USENIX Association.

[19] Q. F. J. L. Y. Wang and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *In Proc. of 9th IEEE International Conference on Data Mining (ICDM 09)*, 2009.

[20] Xpolog. http://www.xpolog.com/.

[21] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Online system problem detection by mining patterns of console logs. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM '09, pages 588–597, Washington, DC, USA, 2009. IEEE Computer Society.

[22] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 117–132, New York, NY, USA, 2009. ACM.

[23] J.-G.-L. Q.-F.-S. Y. Y-Xu and J-Li. Mining invariants from console logs for system problem detection. In *In Proc. of 2010 USENIX Annual Technical Conference(ATC 10).*, 2010.

[24] D. Yuan, S. Park, and Y. Zhou. Characterizing logging practices in open-source software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 102–112, Piscataway, NJ, USA, 2012. IEEE Press.