System Problem Detection by Mining Console Logs

Wei Xu

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-112
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-112.html

August 1, 2010

System Problem Detection by Mining Console Logs

by

Wei Xu

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Fall 2010

System Problem Detection by Mining Console Logs

Abstract

System Problem Detection by Mining Console Logs

by

Wei Xu

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor David A. Patterson, Chair

The console logs generated by an application contain information that the developers believed would be useful in debugging or monitoring the application. Despite the ubiquity and large size of these logs, they are rarely exploited because they are not readily machine-parsable. We propose a fully automatic methodology for mining console logs using a combination of program analysis, information retrieval, data mining, and machine learning techniques. We use source code analysis to understand the structures from the console logs. We then extract features, such as execution traces, from logs and use data mining and machine learning methods to detect problems. We also use a decision tree to distill the detection results to a format readily understandable by operators who need not be familiar with the anomaly detection algorithms. The whole process requires no human intervention and can scale to large scale log data. We extend the methods to perform online analysis on console log streams. We evaluate the technique on several real-world systems and detected problems that are insightful to systems operators.

To my family

Contents

List of Figures

List of Tables

Acknowledgments

I am grateful for the collaboration, guidance and help during my entire PhD career.

First and foremost, I am extremely fortunate to have two great co-advisors, Professor David Patterson and Professor Armando Fox. Throughout my career as a Ph.d. student at Berkeley, they provided incredible help on every aspect of the research. From choosing a research topic to further shaping the project, from technical directions to fixing English problems, their help penetrates everywhere in this dissertation. I am especially grateful for their encouragement and advice during the time when I had trouble finding a research topic. I feel very fortunate to have the opportunity to learn from them.

My special thanks go to Ling Huang at Intel. We collaborated on every detail in the project, and he captures even the slightest flaws existed and his insights of both machine learning and systems shaped the entire project.

I also appreciate Professor Michael Jordan, who taught me about machine learning and helped me solving many problems I had during the project.

My thanks go to my qualifying exam committee, Professor Eric Brewer, Professor Pieter Abbeel and Professor Ray Larson for their great suggestions that helped improving the dissertation, and especially for their encouragement. Professor Pieter Abbeel and Professor Ray Larson are also on my dissertation committee, helping me further improve the dissertation.

I enjoyed the supportive and intellectually stimulating research environment of the RAD Lab. All RAD Lab faculty, students and retreat guests provided important input throughout this research.

Getting real data from industry is an essential part for this research. I appreciate Urs Hoelzle's help for allowing me to use Google's log data. At Google, a number of kind managers and engineers helped me reviewing and explaining the data, especially Jay Sutaria, Alex Wu, Lea Kissner devoted a great amount of effort into the process. I also want to thank Bill Bolosky and Devendra Jaisinghani for providing log data from Microsoft Research and EBay.

I am also fortunate to get help from many industry experts for their invaluable comments on various papers and presentations related to the dissertation: Joe L. Hellerstein, Urs Hoelzle and Deborah Weisser from Google, Feng Zhao, Bill Bolosky, Richard Draves, Alice Zheng and Moises Goldszmidt from Microsoft Research, Byung-Gon Chun, Jaideep Chandrashekar and Petros Maniatis from Intel Labs, Peter Vosshall from Amazon Web Services, Kimberly Keeton, Xiaoyun Zhu, Zhikui Wang and Sharad Singhal from HP Labs, Bill Kramer and John Shalf from Lawrence Berkeley National Lab, and Jon Stearley from Sandia National Lab.

When I was aimless in finding a research topic, Feng Zhou and Li Zhuang kindly offered me an internship position at Netease Yodao in Beijing, where I, for the first time, built a real system and found an exciting research opportunity that leads to this dissertation.

I also express my gratitude to my family – my father, mother and especially my wife have always been so supportive during this research.

This research is supported in part by gifts from Sun Microsystems, Google, Microsoft, Amazon Web Services, Cisco Systems, Cloudera, eBay, Facebook, Fujitsu, Hewlett-Packard, Intel, Network Appliance, SAP, VMWare and Yahoo! and by matching funds from the State of California's MICRO program (grants 06-152, 07-010, 06-148, 07-012, 06-146, 07-009, 06-147, 07-013, 06-149, 06-150, and 07-008), the National Science Foundation (grant #CNS-0509559), and the University of California Industry/University Cooperative Research Program (UC Discovery) grant COM07-10240.

Chapter 1

Introduction

When a datacenter-scale service consisting of hundreds of software components running on thousands of computers misbehaves, developer-operators need every tool at their disposal to troubleshoot and diagnose operational problems. Ironically, there is one source of information that is built into almost every piece of software that provides detailed information that reflects the original developers' ideas about noteworthy or unusual events, but is typically ignored: the humble console log.

In this project, we use a combination of program analysis, information retrieval and machine learning techniques to automatically extract structured information out of console logs, and automatically detect and visualize runtime problems in large scale distributed systems. We applied our methodology to a variety of real-world systems and got insightful detection results.

We first discuss the difficulties facing current system monitoring solutions, and the advantages/disadvantages of using free text console logs. We then summarize the methodology we used for log analysis and our contributions. Finally, we give a quick preview of the detection results on a number of case studies.

1.1 Background and Motivation

Monitoring and diagnosing large scale systems is hard

Today's large-scale Internet services run in large server clusters. A recent trend is to run these services on virtualized cloud computing environments such as Amazon's Elastic Compute Cloud (EC2) [3]. These system architectures enable highly scalable Internet services at a relatively low cost. However, solving problems in such systems brings new challenges for both system developers and operators. One significant problem is that as the system scales, the amount of information operators need to process goes far beyond the level that can be handled manually, and thus there is a huge demand for automatic processing of monitoring data.

Much work has been done on automatic problem detection and diagnosis in such systems. Researchers and operators have been using all kinds of monitoring data, from the simplest numerical metrics such as resource utilization counts [58, 19, 8] to system events [45, 67] to more detailed tracing such as execution paths [17, 16]. Sometimes operators even dedicate a significant portion of application code for monitoring [63]. Metrics collected provide different levels of insights into the systems under study and can often help operators and developers solve problems in their systems, we introduce these methods in detail in Chapter 7.

Monitoring infrastructures are costly to maintain in an Internet service system. These systems integrate external (often open source) components. These components may not have the required instrumentation built-in and thus require programmer overhead to implement. Even worse, these components are frequently revised or upgraded, keeping the instrumentation with this churn rate makes the overhead even more significant. Also, as developers cannot always predict what kind of problems that a particular set of metrics can help diagnose, it is often hard to justify such instrumentation to collect them.

On the other hand, console logs are original instrumentation built into almost every software systems by the developers. The overhead for collecting console logs are very low, but the information contained in console logs can be rich. In this project, we develop a general technique to automatically discover the information contained in console logs that can be used to supplement or even replace custom instrumentations.

Console logs are everywhere, but not commonly used

Since the dawn of programming, developers have used everything from *printf* to complex logging and monitoring libraries [31, 39] to record program variable values, trace execution, report runtime statistics, and even printing out full-sentence messages designed to be read by a human—usually by the developer. However, modern large-scale services usually combine large open-source components authored by hundreds of developers, and the people scouring the logs—part integrator, part developer, part operator, and charged with fixing the problem—are usually not the people who chose what to log or why. (We'll use the term operator to represent a potentially diverse set of people trying to detect operational problems.) Furthermore, even in well-tested code, many operational problems are dependent on the deployment and runtime environment and cannot be easily reproduced by the developer. Thus, it is unavoidable that people other than the original developers need to source logs from time to time when diagnosing problems. Our goal is to provide them with better tools to extract value from the console logs.

As logs are too large to examine manually [50, 75] and too unstructured to analyze automatically, operators typically create *ad hoc* scripts to search for keywords such as "error" or "critical," but this has been shown to be insufficient for determining problems [50, 75]. Rule-based processing [80] is an improvement, but the operators' lack of detailed knowledge about specific components and their interactions makes it difficult to write rules that pick out the most *relevant* sets of events for problem detection. To make things worse, high code churn rates in Internet service systems make manually specified rules infeasible to maintain overtime, as code updates can change what's in the logs and the relevance of certain messages. Instead of asking users to search, we provide tools to automatically find "interesting" log messages.

1.2 Contributions

We aim to build a fully automatic problem detection system using only console log information. Our goal is to find the "needles in the haystack" that might indicate operational problems, without any manual input.

Since unusual log messages often indicate the source of the problem, it is natural to formalize log analysis as an *anomaly detection* problem in machine learning. However, it is not always the case that the presence, absence, or frequency of a single type of message is sufficient to pinpoint the problem; more often, a problem manifests as an abnormality in the relationships among different types of log messages as correlations, relative frequencies, and so on. Therefore, instead of analyzing the words in textual logs (as done, for example, in [89]), we create features that accurately capture various correlations among log messages, and perform anomaly detection on these features. Constructing such features requires parsing console logs accurately to extract information embedded in the free text messages, such as variable values. Existing log parsing methods (discussed in later chapters) do not provide the required accuracy, so we designed a new log parser leveraging program source code information, which is part of our contribution.

We studied logs and source code of many popular software systems used in Internet services, and observed that a typical console log is much more structured than it appears: the definition of its "schema" is implicit in the log printing statements, which can be recovered from program source code. This observation is key to our log parsing approach, which yields detailed and accurate features. Given the ubiquitous presence of open-source software in many Internet systems, we believe the need for source code is not a practical drawback to our approach. In cases when source code is not readily available or too hard to manage, we show that we can extract the "schema" of log messages directly from program binaries without much sacrifice on accuracy.

We make the following the three contributions in this dissertation.

1. A general methodology for automated console log processing

Our first contribution is a general four-step methodology that allows machine learning and information retrieval techniques to be applied to free-text logs. Our approach requires no changes to existing software and works on existing textual console logs of any size. Specifically, our methodology involves the following four aspects:

1. A technique for analyzing source code or program binaries to recover the structure inherent in console logs;

2. The identification of common information in logs—state variables and object identifiers—and the automatic creation of features from the logs (exploiting the structure found) that can be subjected to analysis by a variety of machine learning algorithms;

3. Demonstration of a machine learning and information retrieval methodology that effectively detects unusual patterns or anomalies across large collections of such features extracted from a console log;

4. Where appropriate, automatic construction of a visualization that distills the results of anomaly detection in a compact and operator-friendly format that assumes no understanding of the details of the algorithms used to analyze the features.

2. Online problem detection with message sequences.

Our second contribution is a novel two-stage online log processing approach that combines frequent pattern mining with Principal Component Analysis (PCA) based anomaly detection to allow fast and accurate detection on features based on sequences of messages. In particular, we show how to trade off time-to-detection vs. accuracy in the online setting by augmenting frequent-sequence information with timestamp information. We show that we can achieve similar detection accuracy as the offline approach. As a beneficial side effect, the pattern mining aspect of our approach can potentially help operators better understand system behavior even under normal conditions.

3. System implementation and evaluation on real world systems.

Our third contribution is the implementation of the log processing system and evaluations on real world systems. We implemented parsers to extract message templates from source code written in a variety of programming languages, including C, C++ (Macros heavy), Java (Object-oriented), and even scripting languages like Python. In cases where source code is difficult to obtain or manage, we showed that we can achieve similar parsing accuracy by analyzing program binaries for Java byte code and binaries compiled from C codes.

We implemented both of our parsing and feature extraction steps in an "embarrassingly parallel"[1] style, allowing us to run them as Hadoop [9] map-reduce jobs using cloud computing, achieving nearly linear speedup for a few dollars per run.

We evaluate our approach and demonstrate its capability and scalability with three real-world systems: the Darkstar online game server [90], the Hadoop File System [9] and a production system at Google. In all these three systems, we showed that our log analysis method not only helps system operators discover operational problems, but also provides insights to developers about potential bugs. For example, in the Darkstar case, our method accurately detects performance anomalies immediately after they happen and provides hints as to the root cause. As another example, in a 203-node Hadoop cluster, we detect runtime anomalies that are commonly overlooked, and distill over 24 million lines of console logs to a one-page decision tree that a domain expert who is not familiar with machine learning can readily understand. Studying the detection results, we are able to find a number of temporary failures affecting performance, a bug causing data loss, as well as bad log messages confusing many hadoop users. We also focus on scalability of our method. The analysis of 24 million lines of Hadoop logs can be done with Hadoop map-reduce on 60 Amazon EC2 nodes within 3 minutes. Google system is a production cluster consisting of thousands of nodes. Comparing to Darkstar and Hadoop logs, The Google data set is five magnitudes larger, and it is also more complex in terms of the amount of information contained. Despite of these differences, we can apply almost same log analysis methods and

[1]The parallel is "embarrassing" because the workload separates into different independent tasks without effort. In our case, parsing one log message does not depend on other messages, once we obtain all message templates.

get insightful results. All these examples demonstrate the generality and applicability of our log analysis approach to a variety of real world systems.

In summary, the combination of elements in our approach, including our novel combination of source code analysis with log analysis and automatic creation of features for anomaly detection, enables a level of detail in log analysis that was previously impossible due to the inability of previous methods to correctly identify the features necessary for problem identification.

1.3 Summary

Despite the fact that console log is a rich information source for system monitoring and diagnosis, it is not fully utilized by system operators today due to its free-text nature. In this dissertation, we present a general approach to automatically process console logs to detect problems in large scale Internet service systems. Our contributions include the methodology design, system implementation, and application to real-world systems.

In the next chapter, we summarize the key insights into the console log analysis problem, and highlight how we turn these insights into a four-step methodology design. Then, we summarize a number of systems we surveyed to demonstrate the generality of our approach. We also introduce three data sets, which contain from a million to hundreds of billions of lines of console logs, which we use in the evaluation of our methodology throughout this dissertation.

The rest of the dissertation proceeds as follows: Chapter 3 describes our log parsing technique and different parser implementations in detail. Chapter 4 presents two complete case studies in feature creation, anomaly detection, and visualization. Chapter 5 extends the methodology to perform online problem detection on console log streams. We discuss the real world application of this methodology on Google's production data in Chapter 6. In Chapter 7, we review existing work on console log mining, and in system monitoring and diagnosis in general. We suggest possible future directions in Chapter 8 and conclude in Chapter 9.

Chapter 2

Key Insights and Overview

Unlike most existing work that either treats console logs as a collection of English words [87, 80], or as a time series of events [60, 61, 99], our model of console logs is fundamentally different. In Section 2.1, we summarize our insights into console log: console logs can be automatically transformed into a number of interleaving event sequences, based on fine-grained information extracted from free text messages. These event sequences best represents program executions and thus the best way to detect problems.

We provide an overview our methodology of automatically analyzing console logs in Section 2.1 and highlight the key components in our systems and their importance in the entire framework.

In order to evaluate the generality of our approach, we surveyed console logs from a number of real-world systems, ranging from operating system kernels to web applications. Section 2.3 shows our methodology applies to most of them. We also provide detailed description of three sets of console logs we use as case studies throughout the dissertation.

2.1 Key Insights

Important information is buried in the millions of lines of free-text console logs. To analyze logs automatically, we need to create high quality *features*, the numerical representation of log information that is understandable by a machine learning algorithm. The following four key observations lead to our solution to this problem.

Source code is the "schema" of logs.

Although console logs appear in free text form, they are in fact quite structured because they are generated entirely from a relatively small set of log printing statements in the system.

Consider the simple console log excerpt and the source code that generated it in Figure 2.1. Intuitively, it is easier to recover the log's hidden "schema" using the source code

```
starting: xact 325 is COMMITTING
starting: xact 346 is ABORTING
```

```
CLog.info("starting: " + txn);
Class Transaction {
  public String toString() {
    return "xact " + this.tid +
      " is " + this.state;
  }
}
```

Figure 2.1: Top: two lines from a simple console log. Bottom: Java code that could produce these lines.

information (especially for a machine). Our method leverages source code analysis to recover the structure of logs.

The most significant advantage of our approach is that we are able to accurately parse *all possible* log messages, even the ones rarely seen in actual logs. In addition, we are able to eliminate most heuristics and guesses for log parsing used by existing solutions.

Common log structures lead to useful features

A person usually reads the log messages in Figure 2.1 as a constant part (`starting: xact ... is`) and multiple variable parts (`325`/`326` and `COMMITTING`/`ABORTING`). In this dissertation, we call the constant part the *message type* and the variable part the *message variable*.

Message types – marked by constant strings in a log message – are essential for analyzing console logs and have been widely used in earlier work [61]. In our analysis, we use the constant strings solely as markers for the message types, completely ignoring their semantics as English words, which are known to be ambiguous [75].

Message variables carry crucial information as well. In contrast to prior work that focuses on numerical variables [61, 75, 99], we identified two important types of message variables for problem detection by studying logs from many systems and by interviewing Internet service developers / operators who heavily use console logs.

Identifiers are variables used to identify an object manipulated by the program (e.g., the transaction ids `325` and `346` in Figure 2.1), while *state variables* are labels that enumerate a set of possible states an object could have in program (e.g. `COMMITTING` and `ABORTING` in Figure 2.1). Table 2.1 provides extra examples of such variables. We can determine whether a given variable is an identifier or a state variable progmatically based on its frequency in console logs. Intuitively, state variables have a small number of distinct values while identifiers take a large number of distinct values (detailed in Section 4.1).

We acknowledge that logs also contain other types of message variables such as times-

Variable	**Examples**	**Distinct values**

Identifiers	`transaction_id` in Darkstar;
	`block_id` in Hadoop file system;
	`cache_key` in Apache HTTP server;
	`task_id` in Hadoop map reduce.
		many

State Vars	Transaction stages in Darkstar;
	Server names in Hadoop;
	HTTP status code (`200`, `404`);
	POSIX process return values.
		few

Table 2.1: State variables and identifiers

tamps and various counts. We do not discuss those variables in this paper as they have been well studied in existing work [89].

Message types and variables contain important runtime information useful to the operators. However, lacking tools to extract these structures, operators either ignore them, or spend hours *grep'ing* and manually examining log messages, which is tedious and inefficient.

Our accurate log parsing allows us to use structured information such as message types and variables to automatically create features that capture information conveyed in logs. To our knowledge, this is the first work extracting information at this fine level of granularity from console logs.

Message sequences are important in problem detection

When log messages are grouped properly into message sequences, there is a strong and stable correlation among messages within the same group. For example, messages containing a certain file name are likely to be highly correlated because they are likely to come from logically related execution steps in the program.

A message sequence is often a better indicator of problems than individual messages. Many anomalies are only indicated by incomplete message sequences. For example, if a write operation to a file fails silently (perhaps because the developers do not handle the error correctly), no single error message is likely to indicate the failure. By correlating messages about the same file, however, we can detect such cases by observing that the expected "closing file" message is missing.

Previous work grouped logs by time windows only, and the detection accuracy suffers from noise in the correlation [50, 89, 99]. In contrast, we create message groups based on more accurate information, such as the message variables described above. In this way, the correlation is much stronger and more readily encoded so that the abnormal correlations also become easier to detect.

Although we could also adopt traditional log mining method, we focus on features based on message sequences in this dissertation. Of course, sequence-based features bring additional challenges in an online detection setting, as we have to decide when detection can

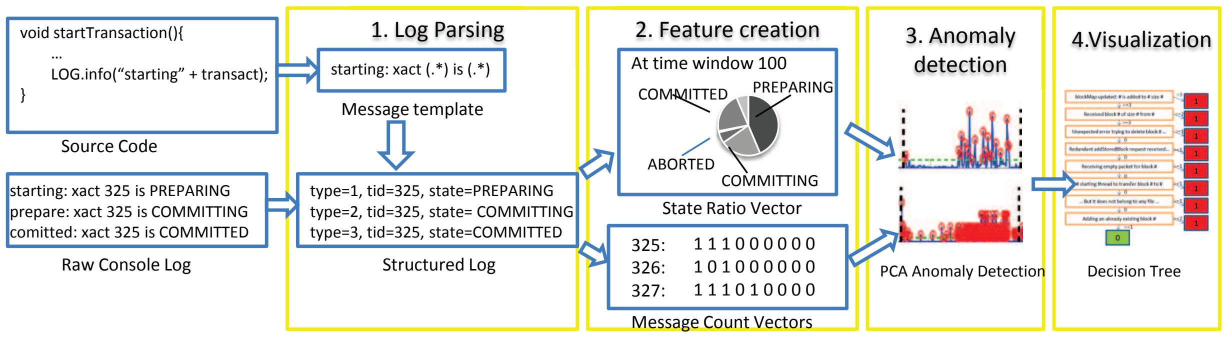


Figure 2.2: Overview of four-step console log analysis methodology.

begin a continuously growing sequence. We solve this problem using frequent-pattern-based filtering techniques.

Strong patterns with lots of noise

Our last important observation in production console logs is that the *normal* patterns - whether in terms of frequent individual messages or frequent message sequences - are very obvious. It is obvious why these strong patterns exist: in production systems, most of the operations are normal, and generate normal log sequences. This observation enables us to use simple machine learning algorithms, such as frequent pattern mining and Principal Component Analysis (PCA), for problem detection.

On the other hand, due to the console log generation and collection process, much noise is introduced. The two most notable kinds of noise are the random interleaving of messages from multiple threads or processes as well as the inaccuracy of message ordering. Our grouping methods help reduce this noise, but the detection algorithm still needs to tolerate the noise. We discuss the patterns and noise that exist in console logs in Chapter 5, when we introduce our online problem detection techniques.

2.2 Methodology Overview

Figure 2.2 shows the four steps in our general framework for mining console logs.

Step 1: Log parsing

We first convert a log message from unstructured text to a data structure that shows the message type and a list of message variables in (name, value) pairs. We get all possible log message template strings from either the source code or program binaries and match these templates to each log message to recover its structure (that is, message type and variables). We discuss our parsing method and implementations for different programming languages in Chapter 3. Our experiments show that we can achieve high parsing accuracy in a variety of real-world systems.

There are systems that use structured tracing only, such as BerkeleyDB (Java edition). In this case, because logs are already structured, we can skip this first step and directly apply our feature creation and anomaly detection methods. Note that these structured logs still contain both identifiers and state variables.[1]

Step 2: Feature creation

Next, we construct feature vectors from the extracted information by choosing appropriate variables and grouping related messages. Our method provides flexibility for generating features, including application-specific ones that incorporate operators' domain knowledge. The structured parsing step makes it very easy to implement and parallelize the feature creation algorithms.

In this dissertation, we first give the construction of two widely applicable features: the state ratio vectors and the message count vectors, both of which are unexploited in prior work. As we will show in Chapter 4, the state ratio vector feature is trivial to use in an online detection setting. In contrast, the message count vector feature is nontrivial to construct over an online log stream. We demonstrate our method for doing so in Chapter 5.

Step 3: Machine learning

Next, we apply machine learning methods to mine feature vectors. In this dissertation, we focus on using anomaly detection techniques, labeling each feature vector as normal or abnormal. We find that the Principal Component Analysis (PCA)-based anomaly detection method [23] works very well with both features. This method is an unsupervised learning algorithm, in which all parameters can be either chosen automatically or tuned easily, eliminating the need for prior input from the operators. In an online setting, we added an extra filtering step, which uses frequent pattern based methods to take care of the vast majority of normal messages quickly. Combining these two methods, we can achieve both small detection latency and high detection accuracy.

Although we only demonstrated two specific machine learning algorithms in this dissertation, neither is intrinsic to our approach, and different algorithms utilizing different extracted features could be readily "plugged in" to our framework.

Step 4: Visualization.

Finally, in order to let system integrators and operators better understand the PCA anomaly detection results, we visualize results in a *decision tree* [98]. Compared to the

[1]In fact, the last system in Table 2.3 (Storage Prototype) is an anonymous research prototype with built-in customized structured traces. Without any context, even without knowing the functionality of the system, our feature creation and anomaly detection algorithm successfully discovered log segments that the developer found insightful.

PCA-based detector, the decision tree provides a more detailed explanation of how the problems are detected, in a form that resembles the event processing rules [42] with which system integrators and operators are familiar.

The four steps discussed above can either work as a coherent system, or be applied individually to certain data sets as required. For example, on tracing data that are already structured, we can directly apply the feature extraction step without parsing. Users can also add log parsers for specific programming languages, create application specific features, apply good machine learning algorithms for problem detection, etc.

In particular, our framework follows a modularized implementation: Each step is implemented in one or multiple modules, and the interfaces between modules are simple and well defined. Thus it is easy to add or replace an implementation for each step.

2.3 Case Studies and Overview of Results

A Survey of console logs

We studied source code and logs from a total of 29 different systems. Twenty-five of them are widely deployed open source systems, one is a research prototype, one is a proprietary web application's Flash client, and the last three are production systems at Google. Table 2.3 summarizes the results. Though these systems are distinct in functionality, developed using different languages by different developers at different times. 27 of the 29 systems use free text logs, and our source-code-analysis based log parsing applies to all of these 27.

Though these systems cover several popular programming languages, including C, C++, Java, Python and ActionScript3, their logs have many common properties. For example, we found that about 1%-5% of code lines are logging calls in most of the systems, but most of these calls are rarely, if ever, executed because they represent erroneous execution paths. It is almost impossible to maintain log-parsing rules manually with such a large number of distinct logger calls, which highlights our advantage of discovering message types automatically from source code. On average, a log message reports a single variable. However, there are many messages such as `starting server` that report no variables, while other messages can report 10 or more. Also, we can find at least one state variables or identifiers in 28 of the 29 systems in Table 2.3 (22 have both), confirming our assumption of their prevalence.

Of course, different programming languages have different logging styles. Generally speaking, there are two styles: using format strings and using string concatenation. Most C programs use *printf* style format strings for logging. C++ programs use both string concatenation style *cout* streams and *printf* style formatting strings. It is common practice for programmers to use wrapper functions or macros to generate standard information such as time stamps and severity levels. These wrappers can sometimes be detected automatically from the format string parameter. However, as we will discuss in Chapter 3, some wrappers add extra variables into the message, which requires more detailed analysis.

Java and Python programs usually use string concatenation to generate log messages and often rely on standard logger packages (such as *log4j* or its Python ports). Analyzing

System	Nodes	Messages	Log Size
Darkstar	1	1,640,985	266 MB
Hadoop (HDFS)	203	24,396,061	2412 MB
Google System1	tens of thousands [2]	tens of billions	400 GB

Table 2.2: Data sets used in evaluation. Nodes=Number of nodes in the experiments.

these logging calls requires understanding data types, which we detail in Chapter 3. Our source-code-analysis based log parsing approach successfully works on most of them.

Data set used in this dissertation

To be succinct yet reveal important issues in console log mining, we focus further discussion on three representative systems shown in Table 2.3: the Darkstar online game server, the Hadoop File System (HDFS), and a production system used at Google. All these systems handle persistence, an important yet complicated function in large-scale Internet services. However, they are different in nature. Darkstar focuses on small, time sensitive transactions, while HDFS is a file system designed for storing large files and batch processing. Google's system is also a storage system, but runs at much larger scale. Darkstar and Hadoop are both written in Java, and represent two major open source contributors (Sun and Apache, respectively) with different coding and logging styles. Google's system is implemented in C++ and has its own coding standards and logging libraries.

For HDFS and Darkstar data, we collected logs from systems running on Amazon's Elastic Compute Cloud (EC2) and we also used EC2 to analyze these logs. Table 2.2 summarizes the log data sets we used. The Darkstar example revealed a behavior that strongly depended on the deployment environment, which led to problems when migrating from traditional server farms to clouds. In particular, we found that Darkstar did not gracefully handle performance variations that are common in the cloud-computing environment. By analyzing console logs, we found the reason for this problem, as discussed in detail in Section 4.3.1.

Satisfied with Darkstar results, to further evaluate our method we analyzed HDFS logs, which are much more complex. We collected HDFS logs from a Hadoop cluster running on over 200 EC2 nodes, yielding 24 million lines of logs. We successfully extracted log segments indicating run-time performance problems that have been confirmed by Hadoop developers.

The Google data is collected from one of Google's production clusters. It contains logs from each node, ranging from several days to a couple of months, depending on the rate at which messages are generated and the utilization of hard drive space on each particular node. In this dissertation, we focused on logs from a storage system during a two-month period of time. We applied both identifier based detection and state variable based detection techniques. We found that the our analysis methods remained almost the same, despite of size and complexity of the log, and the analysis results were insightful.

All log data are collected from unmodified off-the-shelf systems. Console logs are written directly to local disks on each node and collected offline by simply copying log files, which shows the convenience (no instrumentation or configuration) of our log mining approach. In the HDFS experiment, we used the default logging level, while in the Darkstar experiment, we turned on debug logging (`FINER` level in the logging framework). Google's logging level defaults to a level similar to INFO level in HDFS, but instead of recording each request, it focuses only on important operations such as object creation and background processes.

2.4 Summary

Our log analysis techniques are based on the following four key insights:

- Despite the free-text appearance of console log messages, they are in fact structured, and the source code contains the schema definition of these messages.
- Console logs in many systems contain common information, most notably, identifiers and state variables. This information leads to useful features in problem detection.
- Many important problems in systems are captured not by individual error messages in logs, but by abnormal sequences of messages.
- Console logs from production systems usually exhibit a strong pattern, due to the fact that normal operations dominate in those systems. However, because of the simple mechanism used in console log collection, there can be many inaccuracies.

We developed a four-step methodology for automated analysis of console logs, which includes parsing, feature creation, machine learning and visualization steps. Each step makes the data more structured and less noisy. All four steps are modularized so different algorithms can be "plugged in". In this dissertation, we focus on algorithms generally applicable to many different systems.

To demonstrate the generality of our systems, we surveyed a number of different systems, and show that our methods apply to most of them. We also studied logs from three of these systems in detail. These three systems are different in functionalities, use cases, programming styles, deployment scales, from a single node to thousands of nodes. The logs we collected range from just over a million lines to hundreds of billions of lines. We show that our methods work in this variety of cases.

In the next chapter, we will discuss the source-code-analysis-based log parsing techniques. We introduce parsers implementations for different languages and focus on the language-specific issues in four different languages. We also briefly discuss how we can obtain log structure directly from program binaries when source code is hard to obtain or manage.

System	**Lang**	**Logger**	**Msg Construction**	**Lines of Code**	**Lines of Logs**	**Vars**	**Parse**	**ID**	**ST**
Operating system									
Linux (Ubuntu)	C	custom	printk + printf wrap	7477k	70817	70506	Y	Y[b]	Y
Low level Linux services									
Bootp	C	custom	printf wrap	11k	322	220	Y	N	N
DHCP server	C	custom	printf wrap	23k	540	491	Y	Y[b]	Y
DHCP client	C	custom	printf wrap	5k	239	205	Y	Y[b]	Y
ftpd	C	custom	printf wrap	3k	66	67	Y	Y	N
openssh	C	custom	printf wrap	124k	3341	3290	Y	Y	Y
crond	C	printf	printf wrap	7k	112	131	Y	N	Y
Kerboros 5	C	custom	printf wrap	44k	6261	4971	Y	Y	Y
iptables	C	custom	printf wrap	52k	2341	1528	Y	N	Y
Samba 3	C	custom	printf wrap	566k	8461	6843	Y	Y	Y
Internet service building blocks									
Apache 2	C	custom	printf wrap	312k	4008	2835	Y	Y	Y
mysql	C	custom	printf wrap	714k	5092	5656	Y	Y[b]	Y[b]
postgresql	C	custom	printf wrap	740k	12389	7135	Y	Y[b]	Y[b]
Squid	C	custom	printf wrap	148k	2675	2740	Y	Y	Y
Jetty	Java	log4j	string concatenation	138k	699	667	Y	Y	Y
Lucene	Java	custom	custom log function	217k	143	159	Y[a]	Y	N
BDB (Java)	Java	custom	custom trace	260k	-	-	-	Y	N
Web Applications									
MoinMoin	Python	custom	string replacement	96k	566	611	Y	Y	Y
Trac	Python	custom	string replacement	84k	104	65	Y	Y	Y
AppEngine SDK	Python	custom	string replacement	122k	378	349	Y	Y	Y
Flash Game Client	ActionScript	AS3 built-in	string concatenation	-[d]	-[d]	-[d]	Y	Y	Y
Distributed systems									
Hadoop	Java	custom log4j	string concatenation	173k	911	1300	Y	Y	Y

Continued on next page ...

... Continued									
System	**Lang**	**Logger**	**Msg Construction**	**Lines of Code**	**Lines of Logs**	**Vars**	**Parse**	**ID**	**ST**
Darkstar	Java	jdk-log	Java format string	90k	578	658	Y	Y[b]	Y[b]
Nutch	Java	log4j	string concatenation	64k	507	504	Y	Y	Y
Cassandra	Java	log4j	string concatenation	46k	393	437	Y	N	Y
Storage Prototype	C	custom	custom trace	-[c]	-[c]	-[c]	-[c]	Y	Y
Google System 1	C++	custom	string concatenation	-[d]	10k	-[d]	Y	Y	Y
Google System 2	C++	custom	string concatenation	-[d]	21k	-[d]	Y	Y	Y
Google System 3	C++	custom	string concatenation	-[d]	6k	-[d]	Y	Y	Y

Table 2.3: Survey of console logging in popular software systems. LOC = lines of codes in the system. LOL = number of log printing statements. Vars = number of variables reported in log messages. Parse = whether our source analysis based parsing applies. ID = whether identifier variables are reported. ST = whether state variables are reported.

[a]Logger class is not consistent in every module, so we need to manually specify the logger function name for each module.
[b]System prints minimal amount of logs by default, so we need to enable debug logging.
[c]Source code not available, but logs are well structured so manual parsing is easy.
[d]Omitted due to confidentiality issues.

Chapter 3

Console Logs Preprocessing

The convenience and flexibility of console logs come from the support of using free text messages. This flexibility makes the analysis of such logs very difficult. Console log parsing has been the focus of much prior work. There are tools that help extracting "standard fields" such as timestamps, from logs [87, 35]. These tools, however, are not able to extract *message types* and *message variables* (defined in Section 2.1), the essential information used in this research.

We therefore focus on the free text part of a log message, as shown in Figure 3.1. At the top of the figure, human readers would reasonably conclude that `325`, `346`, `COMMITTING`, and `ABORTING` are message variables while the rest are constant strings marking message types. They could then write a regular expression such as `starting: xact (.*) is (.*)` to "templatize" such log messages. We want to automate this process.

Automating log parsing is essential in Internet services where code churn is very high. Figure 3.2 plots the number of new log printing statements added to the source code each month in four Google systems during the past six years. Systems 1, 2, 3 are relatively mature systems, while System 4 is new development. We can see that there are tens or even hundreds of new log printing statements introduced every month independent of the development stage. These new logs are added either to test new functionalities or to debug pre-existing problems. Without an automated message parsing solution, it is difficult to maintain manually-written rules with such a high churn rate.

Much work has been done to automatically extract variables from free text messages. Some projects use heuristics such as matching numbers or IP addresses [99, 61]. The heuristics are not general enough to handle string-valued variables. Others have used more sophisticated data mining methods to discover the structure of the log message. Most of these methods rely on repeating textual patterns [95, 30, 68]. These methods work well on messages types that occur many times in the log, but they cannot handle rare message types that are likely to be related to the runtime problems. Chapter 7 discusses related work in more detail.

As discussed in Section 2.1, it is much easier for a machine to use the source code as the "schema" for console logs. Our log parsing method contains two separate steps: 1) static source code / binary analysis to extract *message templates* for all possible log messages,

```
starting: xact 325 is COMMITTING
starting: xact 346 is ABORTING
```

```
CLog.info("starting: " + txn);
Class Transaction {
  public String toString() {
    return "xact " + this.tid +
      " is " + this.state;
  }
}
```

Figure 3.1: Top: two lines from a simple console log. Bottom: Java code that could produce these lines. This figure is the same as Figure 2.1 in Chapter 2, and is reproduced here for easy reference.

and 2) runtime log parsing to parse each message. We focus on the accuracy of generated message templates in the first step and on scalability in the second. Notice that the first step is programming language dependent while the second step is not. We implemented parsers for four languages, and they all share the same second step.

In this chapter, we first discuss the "big picture" of our parser design. To be concrete, we provide a running example using our Java parser as a case study in Section 3.2. Then we briefly introduce our parser implementations for C, C++ and Python, highlighting the language specific challenges and solutions. Section 3.4 introduces our methods of extracting message templates directly from program binaries.

We evaluate our parser accuracy with two methods. First, for systems for which we do not have a large enough amount of log data, we use a "micro" evaluation method, in which we compare the message templates with manually extracted templates from program source code. We discuss these "micro" evaluation results along with the introduction of the parser implementations. Second, for systems for which we have enough log data, we perform end-to-end evaluation on both steps of log parsing, and we discuss the results in Section 3.5. Both evaluations demonstrate highly accurate parsing results.

3.1 Console Log Parser Design

3.1.1 Challenges in log parsing

Conceptually, if the software is written in a language like C, it is likely that the template can be directly inferred from *printf* variants that generate the messages, such as

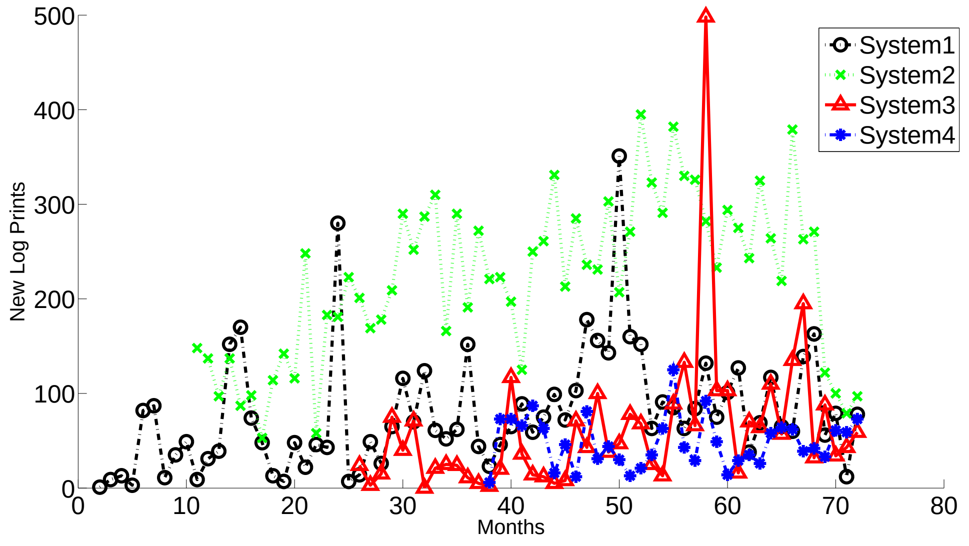


Figure 3.2: Number of new log printing statements added each month in four real Google systems. System 1, 2, 3 are relatively mature systems, while System 4 is new development. We can see that new log messages are constantly introduced at different stages of development.

`fprintf(LOG, "starting: xact %d is %s")`, with the various escapes (`%d`, `%f`, and so on) telling us something about the types of the variables[1].

It is more challenging to handle object-oriented (OO) languages such as Java, which is increasingly used for open source software ([94] and Table 2.3 in Section 2.3). Consider the excerpt of Java source shown in the bottom half of Figure 3.1, which generated the two example log lines. Clearly, the `tid` variable of the `txn` object corresponds to the identifiers 325 and 346 in the log message of Figure 3.1, and the `state` variable corresponds to the labels `COMMITTING` and `ABORTING`. Trying to extract a regular expression by simply “grepping” the source code would only give us `starting: (.*)` (line 1), which does not distinguish `tid` and `state` as separate features with distinct ranges of possible values. Critically, as we will show later, we need this finer level of feature resolution to extract the correct identifier (`tid`) to detect “interesting” problems. Three reasons contribute to this difficulty in OO languages.

1. First, we need to know that `CLog` identifies a logger object; that is, knowing the name of the logger class is not enough.

2. Second, the OO idiom for printing is for an object to implement a `toString()` method that returns a printable representation of itself for interpolation into a string. In this

[1]Technically, the C parsers are complicated by the extensive use of preprocessor macros, which we will detail in Section 3.3.1.

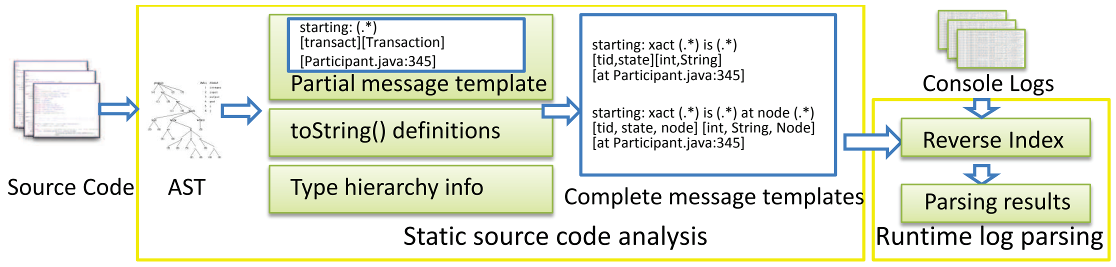


Figure 3.3: Using source code information to parse console logs.

example, the `toString()` method of the abstract type `Transaction` actually reveals the underlying structure of the log message, including the `tid` variable we need for our feature creation algorithm.

3. Third, due to class inheritance, the actual `toString()` method used in a particular call might be defined in a subclass rather than the base class of the logger object.

3.1.2 Parser design overview

All three of the above challenges are addressed by our log parsing method, which consists of two steps: a static source code analysis step and the runtime log parsing step, as Figure 3.3 illustrates.

Step 1: Static source code analysis

The static source code analysis step takes program source (and possibly the names of the logger class) as input. In this step, we first generate the source code's *abstract syntax tree* (AST) [4], a popular data structure for traversing and analyzing source code. We use the AST implementations built into the open-source Eclipse IDE for both Java and C/C++ codes [82, 38], while dynamic scripting languages like Python has built-in functionality to analyze AST [66]. For programs that use printf-style logging, we identify all the logger calls in AST and extract the message template directly from the format string.

Object oriented languages bring extra complexity in the static analysis step, as discussed in the previous section. We still automatically discover all log print statements, but analyzing these statements alone gives us only a *partial* message template, since the template may involve interpolation of objects of non-primitive types, as in line 1 of the source code excerpt in Figure 3.1.

In addition, we do type inference on all objects appearing in the log printing statements. For each type, we then discover its `toString()` definitions, and look at the string formatting statements in those calls to deduce the types of variables in message templates, substituting this type information back into the partial templates. We do this recursively until all templates interpolate only primitive types; if no `toString()` method can be found for a particular variable anywhere along its inheritance path, we assume that that variable can take on any string value and we do no further semantic interpretation. A single pass can accomplish all of these operations over the AST. The output of the process is the set of *complete* message templates, with a data structure containing each message's template (regular expression), position in the source code, and the names and data types of all variables appearing in the message.

We describe a concrete example of implementing a parser for Java in the Section 3.2 and briefly introduce the language-specific challenges in Section 3.3.

Notice that our source code analysis only uses function call statement information and data type information. Sometimes, when source code is not readily available or is too hard to manage, we can extract the similar information directly from program executables. We discuss our implementations of binary parsers for both Java class files and Windows executables compiled from C programs in Section 3.4.

Step 2: Runtime log parsing

In a nutshell, in the runtime log parsing step, we perform the following two operations: finding the matching template for each log message and extracting message variables using the template. Note that finding the matching template for a message implies identifying the message type. The goal is to make the process efficient and scalable.

The most computationally intensive step is finding the matching template, because Table 2.3 in the previous chapter shows that there can be tens of thousands of message templates, and we need to do the matching for each message. In order to make the process of matching template fast, we first compile all message templates into an Apache Lucene [44] reverse index [69], which allows us to quickly associate any log message with the corresponding template. The index fits in memory easily (it is a few MB in size), and once the index is loaded, for each log message, the matching process goes as following.

1. Using established heuristics in log analysis [61, 92], we construct an index query from each log message by removing all numbers and special symbols.

2. We search the index with the query constructed, and rank results by relevance score.

3. From the list of relevance-ranked candidate results returned by the reverse-index search, we pick the highest-ranked result that allows a regular expression match to succeed against the log message.

After finding the matching template, the actual variable extraction is a simple regular expression operation. At this step, we also try to add information such as data type from the source code to the parsing results.

We note that once the reverse index is distributed to each of the parser node, the parsing step is embarrassingly parallel, because parsing of each message is independent of other messages. We implement the parser as a Hadoop map-reduce job by replicating the index to every worker node and partitioning the log among the workers, achieving near linear speedup. The map stage performs the reverse-index search; the reduce stage processing

```
0  Transaction transact = ...;
1  Log.info("starting: " + transact);


2  Class Transaction {
3    public String toString() {
4      return "xact " + this.tid +
5        " is " + this.state;
6    }
7  }

8  Class SubTransaction extends Transaction{
9     private Node node = ....;
10    public String toString() {
11      return "xact " + this.tid +
12        " is " + this.state +" at "+ node;
13    }
14  }

15 Class TransactExec extends Transaction {
16 .....
```

Figure 3.4: Example source code segments. Notice that the logger call in line 1 may generate different log messages at runtime due to the class hierarchies.

depends on the features to be constructed, and Section 4.1 in the next chapter shows two examples of such features.

In particular, we do not claim to handle every situation correctly (despite extensive support for language idioms), as we are essentially guessing programs' runtime behavior from static analysis. However, we do show that the high accuracy of our parser enable us to extract features previously not possible from existing log parsers.

The most complex part of the techniques is in the static analysis step, which is also programming language specific. In the next two sections, we provide examples of such parsers to illustrate the subtleties in the parser design.

3.2 Object Oriented Languages: A Running Example

Designing a parser to extract message templates for object oriented languages is the most involved. We illustrate the details of our source code analysis techniques for message template extraction with a running example in Java, though the general idea applies to other object-oriented languages as well.

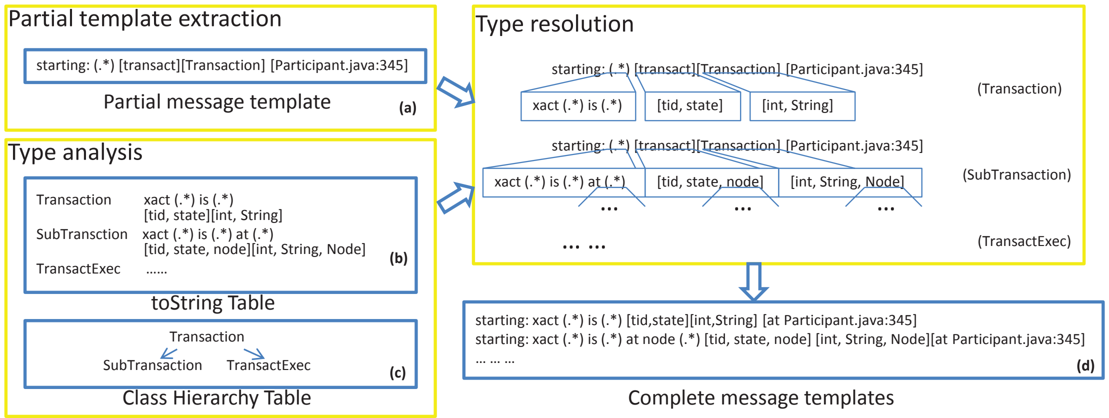

Partial template extraction
starting: (.*) [transact][Transaction] [Participant.java:345]
Partial message template
(a)
Type analysis
Transaction xact (.*) is (.*) [tid, state][int, String]
SubTransction xact (.*) is (.*) at (.*) [tid, state, node][int, String, Node]
TransactExec ......
(b)
toString Table
Transaction
SubTransaction
TransactExec
(c)
Class Hierarchy Table
Type resolution
starting: (.*) [transact][Transaction] [Participant.java:345]
xact (.*) is (.*)
[tid, state]
[int, String]
(Transaction)
xact (.*) is (.*) at (.*)
[tid, state, node]
[int, String, Node]
(SubTransaction)
(TransactExec)
starting: xact (.*) is (.*) [tid,state][int,String] [at Participant.java:345]
starting: xact (.*) is (.*) at node (.*) [tid, state, node] [int, String, Node][at Participant.java:345]
(d)
Complete message templates

Figure 3.5: Constructing message templates from source code analysis.

Figure 3.4 is an extended version of the example shown in Figure 3.1. Line 1 of Figure 3.4 is a simple logger call. However, as we discussed in Section 3.1, it might generate different kinds of messages such as

```
starting:  xact 325 is COMMITTING
starting:  xact 346 is ABORTING at n1:8080
```

This complication is because the variable `transact` is a complex data type with multiple `toString()` definitions (Line 2-15). Our goal is to discover *all possible* message templates that Line 1 could generate, so we need to resolve the type hierarchy of `transact`.

Figure 3.5 illustrates the major steps of our approach. All analysis is done on the *abstract syntax tree* (AST) [4] generated by the Eclipse IDE. Our analysis uses three data structures created from the AST: a list of partial message templates, a table of templates representing `toString()` methods for all declared types (the "toString Table"), and a Class Hierarchy table. Although logically the data structures are independent of each other, our implementation builds them using a single pass over the AST.

Partial message template extraction

We first look for all method invocations on objects of the logger class. These invocations give us the list of all log messages that could possibly be generated, whether they actually appear in the log or not. Common logger class libraries such as *log4j*-based loggers [39] can be automatically detected by examining the library the software uses. Analyzing the parameters of the logger call in Line 1 of Figure 3.4 gives the partial message template shown in Figure 3.5 (a). We also record the names and types of message variables interpolated into the log message (such as `transact` in Line 1 of Figure 3.4), which are crucial for the final type resolution, and the filename and line number of the logger call.

Type analysis

We next determine how each message variable will be rendered as a string in the logger call. For example, because `transact` is of type `Transaction`, we can determine how it would appear in a log message by looking at the `toString()` method of the `Transaction` class. We traverse the AST to build a *toString Table* containing the `toString()` definitions and *toString templates* of all classes. Figure 3.5 (b) shows the toString templates extracted from Lines 2–16 in Figure 3.4.

Due to the importance of class hierarchy information, we do a third traversal on AST to build the *Class Hierarchy table*. Box (c) in Figure 3.5 shows an example.

Type resolution

Finally, for each partial message template containing non-primitive variables (i.e., member of a non-primitive class), we lookup the class's *toString* method and corresponding *toString* templates in the toString Table, and substitute the templates found into the partial message template. For example, for the logger call in Line 1 of Figure 3.4 that references the `transact` object, we lookup the *toString* method of its class (`Transaction`). If the `toString()` method is not explicitly defined in `Transaction` class, we use the Class Hierarchy Table, built from the AST, to attempt to resolve `toString()` in the object's superclasses. We do this recursively until either a `toString()` method is found or we reach the root of the class hierarchy (in Java, the `java.lang.Object` class), in which case we give up and treat the template as an unparsed string `(.*)`.

The sub-classing problem is also handled in this step. We find all descendants of a declared class. If there is a `toString()` method defined in any sub-classes, we generate a message template as if the sub-class is used instead of the declared class. For example, because `SubTransaction` is a sub-class of `Transaction`, we generate a second message template capturing the case when `transact` is actually an instance of `SubTransaction`. We do this for every subclass of `Transaction` known at compile time.

Lastly, note that type resolution is recursive. For example, if an object has class `SubTransaction`, we examine the *toString* method of `SubTransaction` (line 8 of Figure 3.4) and we find that it interpolates a variable `node` of non-primitive type (line 11). We recurse and substitute in the toString template of `Node`. We do this until the type of every variable becomes a primitive type or unparsed strings. We also limit the maximum depth of recursions to deal with recursive type definitions.

Corner cases and bad logging

Because we are using static analysis techniques to predict what the log output will look like at runtime, it is impossible to correctly handle all cases. Examples of such cases include loops and recursive calls. We make our technique robust by allowing it to fall back to unparsed string `(.*)` in such cases. In the real systems we studied, these hard cases rarely occur in log printing and rarely cause problems in practice. There are some language-specific idioms such as `Arrays.deeoToString(array)` (array dumping) in Java, which has an implicit built-in format that uses commas to separate array elements. Our parser recognizes these idioms and handles them as special cases.

Of course, our method relies on programmers to write “good” log messages. For example, if programmers use very general types such as `Object` in Java (very rare in practice), our type resolution step fails because there are too many possibilities. We guard against this by limiting the number of descendants of a class to 100, which is large enough to accommodate all logs we studied but small enough to filter out genuine JDK, AWT, and Swing classes with many subclasses (such as `Object`). Features such as generics and mix-ins in modern OO languages provide the mechanisms usually needed to avoid having to declare an object in a very general class. In addition, some log messages are undecorated, emitting only a variable of some primitive type without any constant label. These messages are usually leftovers from the debugging phase, and we simply ignore these messages. It is not a goal of this dissertation to try to fix bad logging practices, but helping programmers producing more useful logs is an interesting future direction (more details in Chapter 8).

3.3 Parsers for Other Languages

In addition to the Java parser discussed in the previous section, we also implemented parsers for some additional languages to demonstrate the generality of our log parsing method.

3.3.1 C and C++

C and C++ are still among the most widely used programming languages today [94]. As we discussed in Section 3.1, it is conceptually simple to extract message templates from C programs, as most C programmers use *printf* style formatting strings, which explicitly reveal the message template. However, in practice, extracting templates from C programs is complicated by the heavy use of preprocessor definitions (i.e. the macros).

C++ programmers use both *printf* style format strings and C++-stream style string concatenations. Although C++ is an object oriented language, the systems we analyzed in this dissertation do not use the object-oriented features for log printing. Thus, we did not perform type resolution steps as we did for Java.

We use built-in parser for Eclipse C Development Tooling (CDT) [38] to parse both C and C++ code into AST. The method we used to traverse the AST to extract all log printing functions and their parameters. The implementation is very similar to the Java parser.

C/C++ programmers heavily utilize macros [53], which complicate our analysis. These macros are handled by the preprocessor before compiling, and thus are not part of the AST. What is worse, the results of these macro expansions are really determined by external information, such as the preprocessor arguments passed to the compiler at build-time and header files that may be OS-dependent.

We cannot simply ignore these macros, as programmers use macros extensively for logging, and ignoring these macros prevents the resulting program from compiling correctly. Our analyzer could analyze the `makefile` to understand what arguments values are used.

However, we face the cross-compiling issue: unless the system we use for static analysis is exactly the same as the system generating logs, the templates might still be different due to the system-dependent macros.

Instead, our code tries to evaluate the macro with all possible branches, and take the union of all the message templates extracted from each branch. In this case, we produce more possible templates than actually exist, but completeness is our goal. These templates use a small amount of extra space in the index, but do not affect our log parsing accuracy as they never match any log messages during the online log parsing stage.

Inter-dependencies of libraries and packages are also more complex for C and C++ programs. In contrast to Java, it is common for an open source C/C++ to require header files, which could contain macros from the libraries. However, it is a challenge to get all source files that could print logs. We currently use heuristics to detect commonly used libraries, and use full text search on the entire repository for any log messages that does not match any template, in order to find the possible source file. Because of this complication, extracting message templates directly from program binaries sometimes yields better accuracy. We discuss the binary analysis approach in Section 3.4.

System	**LOC**	**Manual**	**Parser**	**Accuracy**
Apache HTTP Server 1.3	125,782	499	448	89.8%
Open SSL	84,734	2,348	2,227	94.8%

Table 3.1: Parsing accuracy of C/C++ parser. LOC is the number of source code lines. We compare the number of correctly extracted message templates by the C/C++ parser with the message templates manually discovered from source code. In the Apache case, we only considered `ap_log_error` logger function.

We evaluate the parser with two widely used systems, Apache HTTP Server and OpenSSL library, and Table 3.1 summarizes the results. The relatively lower extraction rate in the Apache case is mainly due to the complexity of the macros used in Apache source code. Specifically, there are nested macros which might need multiple parameters to expand. Our current parser does not handle such cases, so it missed some templates.

Although we could fix these problems, the work in binary parsing makes us aware that most of these logger calls embedded in complex macros are not compiled into the final executables. Thus, we did not improve the parser further along this direction, but instead, we used binary parsing to solve the problem. Section 3.4.2 further discusses the parsing results using program binaries.

3.3.2 Scripting languages (Python)

Python is a very popular scripting language, and it is widely used in Internet service systems to implement both maintenance scripts and user-facing web applications [66]. In this dissertation, we use Python as an example to demonstrate our techniques on scripting languages.

System	LOC	Manual	Parser	Accuracy
MoinMoin	96,363	576	566	98.3%
Trac	84,467	107	104	97.2%
Google App Engine	122,549	382	378	99.0%

Table 3.2: Python parser accuracy. LOC is the number of source code lines. We compare the number of correctly extracted message templates by the Python parser with the message templates manually discovered from source code.

Although it is a scripting language, the Python 2.x's runtime environment provides a built-in *compiler* module, which allows programmers to access AST tree information using normal Python scripts. Like all other languages previously discussed, our Python analyzer is based on traversing the AST to detect log printing statements. Python programs usually use two logging styles, the built-in `print` statements and the standard logging library, which is quite similar to the Java logger [66].

For the purpose of extracting message templates, there are two major differences between a scripting language and conventional languages discussed above. First, variable types are not declared explicitly, but dynamically inferred by the interpreter at runtime. Variable types can even change while the program is running. Dynamic typing makes it hard to implement type inference as we did in the Java case. Fortunately, Python programmers often use a printf-like logging style, specifying the entire format in the log printing statements. In systems we studied, missing type information does not have a major impact on our parsing accuracy.

Second, the syntax is much more flexible than other languages. The most notable idioms are the Perl-style "do or die" expressions. These expressions are commonly used in logging calls. For example, the call `logging.info( "Connection was made from connection %s" %(new_connection() or die()) )` combines logging, the function call, and error handling (or die) in the same statement. Without debating whether this is good programming style, our parser is tuned specifically to accommodate these common constructs. Of course, there are some expressions that we could not handle correctly, as discussed below.

We tested our parser on three widely deployed Python programs: the MoinMoin Wiki [73], the Trac Project Management Tool [26], and Google App Engine Python SDK[2] [37]. These applications cover both common use cases for Python: building web applications (MoinMoin and Trac) and writing operational scripts (App Engine SDK).

Table 3.2 shows that in all these three systems, we can get very accurate parsing results. In fact, there is only one case that our parser does not handle correctly, the *list comprehension* feature. This feature in Python allows programmers to dump the entire list as a variable, so programmers can write lines like `print [c for c in story.comment_set]`. Without proper type information, it is impossible to infer what the list will look like in the printouts.

[2]The SDK is not a server program. It is a tool kit allowing programmers to emulate the Google App Engine Python environment for software development.

3.4 Extracting Message Templates from Program Binaries

Although we described our log parsing method as “source-code-analysis-based”, the technique can be easily extended to use program binaries instead of source code. Obviously, using binary is the only choice if source code is not available, but there are other benefits as well. First, binary is much easier to manage by system operators. Checking-out and building source code are traditionally only performed by developers and may be beyond operators’ skill sets. Second, as we discussed in the C/C++ parsing case, it is often tedious to find source code for all required libraries/modules, especially when these modules are all constantly changing.

Of course, we may not always always get the same results with binaries as using source code. Obviously, most of the variable names are lost, unless the program is compiled with extensive debugging information. Also, it becomes more challenging if C++ stream style logging is used. We did not implement parsers for C++ binaries, which could be an interesting future direction.

We briefly introduce our binary analysis implementations for Java byte code and Windows binaries compiled from C programs. Specifically, in Java, we can achieve the same accuracy using source code or binaries. There are more differences between the templates extracted from C source and binaries, and we discuss these differences in Section 3.4.2.

3.4.1 Java byte code

Java program compiles into an intermediate language, the Java byte code, a special instruction set executed by the Java Virtual Machine (JVM) [62]. Java byte code conceptually represents the instruction set of a stack-oriented, capability architecture. Each byte code opcode is one byte in length, although some require parameters, resulting in some multi-byte instructions. The intermediate byte code representation is a crucial component to support the cross-platform functionality of Java. The byte code is stored in *class files*, and each class file represents a Java class. In addition to the byte code instructions, it also contains meta information such as class name, super classes / interfaces, access flags, and all constants, fields, and method definitions within the class. We leverage these meta information to provide the type-inference part of our message template extraction.

We use an open source Java byte code analysis and manipulation library, ASM [77]. ASM library is designed to generate, transform and analyze compiled Java class files. ASM allows us to use higher level concepts than bytes, such as numeric constants, strings, Java identifiers, Java types, fields, local variables etc [11].

Unlike the AST model we used in the source code analysis case, ASM uses an *event-based* representation of a class, and notifies our analysis code of each instruction or definition found in the class file [11]. Compared to the AST model, which is a top-down approach, the event model parses the program bottom-up. For this reason, we keep a stack in our analyzer to track previous parser events and look for potential function calls and return statements of the `toString()` methods. We find that we can obtain many names of types, functions and class variables from Java byte code, even if the debug option is turned off during compilation. Of course, we do not get names for local variables, but this does not affect the analysis much.

As with Java source code, we scan the byte code in two passes. The first pass builds the type hierarchy graph while the second pass extracts all message templates. Once these two passes are done, the third step of constructing complete message templates is exactly the same as the source code case.

We evaluate the byte code analysis on class files for HDFS and Darkstar (described in Section 2.3). These class files are the same version as the program binary. Instead of building the class files from source code ourselves, we download the binary directly from the project websites to ensure that it is compiled with the "default" settings. The result is very promising: we recover exactly the same set of message templates from each system, except for the local variables and some source code line numbers. This experiment shows that since we do not rely on the variable names for analysis, we get the same result with both binary analysis and source code analysis for the Java case.

3.4.2 Binaries from C programs

Many server programs are still written in C, especially the packages bundled in Linux distributions. Although most of these packages are open source, managing the source code can be overwhelming for system operators. Modern packaging tools such as Debian's dpkg and RPM Package Manager [85] make the effort of making the source code distribution easier, but they do not eliminate the difficulty of building a package from source, which is intimidating to system operators. Thus, we want to support extracting message templates directly from program binaries compiled from C programs.

Conceptually, extracting message template from binaries is simple thanks to the *printf* style format strings. On most platforms, these *printf* strings appear in either data or text segments in program binaries. Also, as there are many log printing statements throughout the program, it is easy to automatically detect which functions often use these strings as parameters, which helps us distinguish the message template strings from other string constants used in the program. Note that it is also correct to include more strings than necessary into the message template index. Those non-template strings are unlikely to match any log message, causing only minor performance hit, without affecting parsing accuracy. Therefore, in our current implementation, we do not make much effort to distinguish whether a string is a printf formatting string or not.

In order to handle binary file formats in different operating systems, we build our analyzer on top of Interactive Disassember (IDA) [24], a popular tool for reverse-engineering. In addition to disassemble binary files, IDA also uses many heuristics to distinguish data types from the bytes in binary. It also tries to include as many debugging symbols as possible, which helps us to recover part of the function names from program binaries. IDA supports its own scripting language, in which our analyzer is written. Our experience shows that IDA can successfully recover most of the strings from both Windows executable files (exe), dynamically linked libraries (dll), and Linux binaries on Intel x86.

Our parser scans through the disassembly result for pointers to string variables. Once we find the variable, we find the next `CALL/JMP` instruction to get the name of the callee. We also record all other parameters passed to the function. Finally, we mark the location of the function call. IDA provides convenient functionality to determine the enclosing functions for each location in the text segment, and we use this functionality to get the caller of the logger. After processing the entire binary file, we use heuristics, such as counting the frequency of using `%` in string arguments for each function, to guess which function is related to logging.

System	**Source**	**Binary**	**Manually Found in Binary**
Apache HTTP Server 1.3	448	258	258
Open SSL	2,227	1,631	1,631

Table 3.3: Comparison between C source parser and C binary parser. The binary are cygwin binaries downloaded from cygwin website. The third column is the number of format strings we found by manually examine the disassembled binary.

Table 3.3 summarizes our comparison results between the source code and binary based parsers. We see major differences between the number of message template extracted from source and binaries. We manually examined the disassembly output in order to find the problem (the third column in Table 3.3). Surprisingly, our manual examination confirmed that all printf format strings that appear in the binary files are correctly extracted by our binary analyzer. Further examination of the results reveals the following two problems causing the difference between source and binary results:

1. As we discussed in Section 3.3.1, the GNU code base often contains code for many platforms. On any specific platform (cygwin in our case), only a fraction of the code is actually compiled into the executables.

2. In the Apache case, the build scripts automatically download and apply patches to the source code before compiling. The patches may add, remove, or modify log printing statements. Our source parser does not have access to these patches, so there might be differences between results from source and binaries.

Because of the two problems above, in fact the source code analysis in this case can neither provides the accurate list of format strings, as almost 50% of the strings never get into the binary, nor cover *all* possible cases, as new message templates might be introduced by patches during the build process. In that sense, we believe using binary analysis can even produce better parsing accuracy, unless the ability to extract variable names is highly desirable.

3.5 Evaluation: Accuracy and Scalability

The parsing results presented in the previous section are “micro” evaluations, which only show the accuracy of extracting message templates from source code or binaries. Lacking

System	**Total Log Lines**	**Failed**	**Failed %**
HDFS	24,396,061	29,636	0.121%
Darkstar	1,640,985	35	0.002%

Table 3.4: Parsing accuracy using Java Parser. Parse fails on a message when we cannot find a message template that matches the message and extract message variables.

a large enough log data set for all ten systems, we present the "end-to-end" evaluation of message-level parsing accuracy in this section using the five data sets introduced in Section 2.3. We achieve over 99% parsing accuracy in all five data sets, which is far higher than existing parsers. We also show that our parser can scale from a single node to thousands of nodes and handle extremely large data sets efficiently.

3.5.1 Accuracy

Table 3.4 shows that our log parsing method achieves over 99.8% accuracy on both systems. Specifically, our technique successfully handled rare messages types, even those that appeared only twice in over 24 million messages in HDFS. On the contrary, word-frequency based console log analysis tools, such as SLCT [95], do not recover either of the features we use in this paper. State variables are too common to be separated from constant strings by word frequency only. In addition, these tools ignore all rare messages, which are required to construct message count vectors.

There are only a few message types that our parser fails to handle. Almost all of these messages contain long string variables. These long strings may overwhelm the constant strings we are searching for, preventing reverse index search from finding the correct message template. However, these messages typically appear at the initialization or termination phase of a system (or a subsystem), when the state of the system is dumped to the console. Thus, we did not see any impact of missing these messages on our detection results.

System	**Total Log Lines**	**Failed %**
Google System 1	46×10^9	<0.001%
Google System 2	8×10^9	<0.012%
Google System 3	8×10^9	<0.011%

Table 3.5: Parsing Accuracy using the C/C++ parser on production data from Google.

Table 3.5 summarizes the message parsing accuracy on Google's production logs. We parsed source code for all libraries referenced by each of the systems in order to cover as many message types as possible. The parser output is checked by a heuristic-based checker to ensure the message type and message variables are correctly extracted. The "Failed %" field in the table is calculated from the output of this checker. Failure cases mostly involve complex array dumps (e.g. programmer could use a function to dump an entire array in a single log message), and long string variables, such as a command line with tens

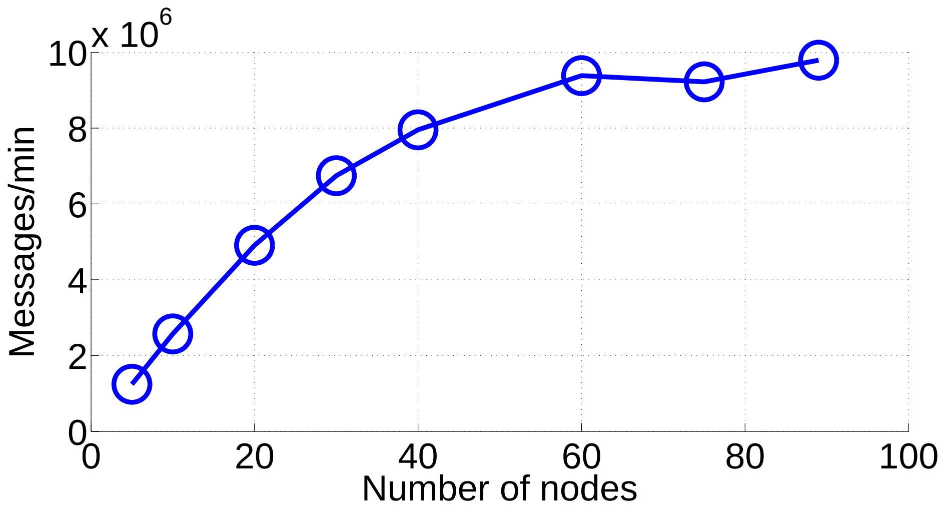


Figure 3.6: Scalability of log parsing with number of nodes used. The x-axis is the number of nodes used, while the y-axis is the number of messages processed per minute. All nodes are Amazon EC2 *high-CPU medium* instances. We used the HDFS data set (described in (Table 2.2) with over 24 million lines. We parsed raw textual logs and generated the message count vector feature (see Section 4.1.2). Each experiment was repeated 4 times and the reported data point is the mean.

of arguments. As these corner cases are relatively rare in the data, we can still achieve high parsing accuracy.

We believe the accuracy of our parsing is essential; only with an accurate parsing system can we extract state variables and identifiers—the basis for our feature construction—from textual logs. Thus, we consider the extra step of static source code / binary analysis to be a small price to pay, given the high quality parsing results that our technique produces.

3.5.2 Scalability

We evaluate the scalability of our log parsing approach with a varying number of EC2 nodes. Figure 3.6 shows the result: Our log parsing and feature extraction algorithms scale almost linearly with up to about 50 nodes. Even though we parse all messages generated by 200 HDFS nodes (with aggressive logging) over 48 hours, log parsing takes less than 3 minutes with 50 nodes, or less than 10 minutes with 10 nodes. When we use more than 60 nodes, the overhead of index dissemination and job scheduling dominate running time.

Google's logs are almost 2000x larger than the data sets used in Figure 3.6. In order to have minimal impact on the production system generating these logs, we limited the resource utilization on each node in log parsing. We were able to distribute the log parsing execution onto thousands of nodes and parse almost two months' worth of logs in less than 8 hours.

Due to the stateless nature of the log parser implementation, all experiments show that our log parsing approach scales up and down easily. We can do the entire log parsing on a single node, or scale to thousands of nodes. We could also perform log parsing as a data stream processor, which is very useful for our online problem detection.

3.6 Summary

Automatic console log parsing is a prerequisite to perform detailed analysis on message types and message variables. Existing log parsing methods suffer from either the tediousness of ad hoc scripting or the lack of accuracy. We used program source code or binary to extract message template and use these message templates to parse logs. Our technique eliminates most of the "guesses" involved in console log parsing. Experiments on eleven real world systems written in four different programming languages show that our technique provides accurate message template extraction and parsing results. Throughout this chapter, we assume no changes to existing logging in the programs, even if there are obvious flaws in logging. We consider an improved logging framework as an interesting future direction (see Chapter 8).

The goal of log parsing is to discover structure within individual messages, turning an unstructured text message into semi-structured form containing message types and message variables. However, as we pointed out in Section 2.1, we need to exploit the inter-message correlations to detect operational problems in these systems. In the next chapter, we discuss our method of grouping related messages, representing these groups as numerical features, and doing anomaly detection.

Chapter 4

Offline Problem Detection and Visualization

Console log parsing makes it easy to handle free textual console logs. However, as we discussed in Chapter 2, being able to parse log messages does not solve the entire problem, as parsing only captures the structure of *single* log message, while most interesting problems in systems can only be discovered by analyzing a *sequence* of related log messages. In this chapter, we introduce the details of the last three steps of our technique: feature creation, anomaly detection and visualization. Throughout this chapter, we assume that the analysis is offline, where we have the access to the entire log trace from a complete execution. Although this offline assumption is sometimes hard to satisfy, it is easier for readers to understand the ideas behind these techniques. We will discuss the techniques for performing online detection in the next chapter.

As we discussed in Chapter 2, we focus on features that are based on correlations among different messages. In this chapter, we discuss two different features, the *state ratio vector* and the *message count vector*, based on state variables and identifiers, respectively. These two features are generally applicable to logs from many different applications, and represent two important types of anomalies in distributed systems: anomaly in aggregated behavior and anomaly in individual operations. These high quality features allow us to apply efficient machine learning techniques and achieve high detection accuracy. We also show that by adopting well known techniques in information retrieval, we can further improve the problem detection accuracy. Finally, because most operators are not familiar with these machine learning techniques, we visualize the detection result in a single-page *decision tree*, which resembles the operation rules that operators are familiar with.

To be concrete in the discussion, we use two real world systems, the Darkstar online game server [90] and Hadoop File System (HDFS) [9] as case studies in this chapter. We present the evaluation results for both case studies in Section 4.3.

4.1 Feature Creation

Console logs contain records on arbitrarily many different aspects of the program execution, from tracing individual operations to reporting the aggregated statistics from the program. The rich variety of data in console logs allows us to extract many *features*, a machine learning term meaning numerical representations of certain aspects of the raw data. Many features are ad-hoc and application specific [61, 75], while some features are common to many different logs. Most existing work models console logs as a single sequence of events and analyzes different message types appearing in time windows [45, 67]. Unlike existing work, we model logs as several interleaving sequences. We also analyze the variables contained in the log messages in addition to message types. We will discuss related work further in Chapter 7.

This section describes our technique for constructing features from parsed logs. We focus on two features, the *state ratio vector* and the *message count vector*, based on state variables and identifiers (see Section 2.1), respectively. The *state ratio vector* is able to capture the aggregated behavior of the system over a time window. The *message count vector* helps detect problems related to individual operations. Both features describe message groups constructed to have strong correlations among their members. The features faithfully capture these correlations, which are often good indicators of runtime problems. Although these features are from the same log, and similar in structure, they are constructed independently, describe different aspects of the system execution, and thus have different semantics.

4.1.1 State variables and state ratio vectors

In many systems, during normal execution the relative frequency of each value of a state variable in a time window usually stays the same. For example, in Darkstar, the ratio between `ABORTING` and `COMMITTING` is very stable during normal execution, but changes significantly when a problem occurs. As another example, HDFS assigns roughly equal number of new blocks to each storage node. The number of blocks assigned to some node might be much smaller than other nodes during a problem period. Notice that the actual number does not matter (as it depends on workload), but the *ratio* among different values matters. State variables can appear in a large portion of log messages. In fact, 32% of the log messages from Hadoop and 28% of messages from Darkstar contain state variables.

We construct the *state ratio vector* $\mathbf{y}$ to encode this correlation: Each state ratio vector represents a group of state variables in a time window, while each dimension of the vector corresponds to a distinct state variable value, and the value of the dimension is how many times this state value appears in the time window.

In creating features based on state variables, we use an automatic procedure that combines two desiderata: 1) message variables should be frequently reported, but 2) they should range across a small constant number of distinct values. The number of distinct values should not depend on the number of messages. Specifically in our experiments, we chose state variables that were reported at least $0.2N$ times, with N the number of messages, and had a number of distinct values not increasing with N for large values of N (e.g., more than a few thousand). Our results were not sensitive to the choice of 0.2.

The time window size is also automatically determined. Currently, we choose a size that allows the variable to appear at least $10D$ times in 80% of all the time windows, where D is the number of distinct values. This choice of time window allows the variable to appear enough times in each window to make the count statistically significant [20] while keeping the time window small enough to capture transient problems. We tried with parameters other than 10 and 80% and we did not see a significant change in detection results.

If a state variable has n distinct values. Recall that each distinct values is represented by a dimension in the feature vector $\mathbf{y}$. Thus, $\mathbf{y}$ is n-dimensional. We stack all n-dimensional $\mathbf{y}$'s from m time windows to construct the $m \times n$ state ratio matrix $\mathbf{Y}^s$.

4.1.2 Identifiers and message count vectors

Identifiers, the message variables used to identify program objects, are also prevalent in logs. For example, almost 50% of messages in HDFS logs contain identifiers. We observe that all log messages reporting the same identifier convey a single piece of information about the identifier. For instance, in HDFS, there are multiple log messages about a block when the block is allocated, written, replicated, or deleted. By grouping these messages, we get the *message count vector*, which is similar to an execution path [31] that would come from custom instrumentation.

To form the message count vector, we first automatically discover identifiers, then group together messages with the same identifier values, and create a vector per group. Each vector dimension corresponds to a different message type, and the value of the dimension tells how many messages of that type appear in the message group.

The structure of this feature is analogous to the *bag of words* model in information retrieval [29]. In our application, the "document" is the message group. The dimensions of the vector consist of the union of all useful message types across all groups (analogous to all possible "terms"), and the value of a dimension is the number of appearances of the corresponding message types in a group (corresponding to "term frequency").

Algorithm 1 summarizes our three-step process for feature construction. We now try to provide intuition behind the design choices in this algorithm.

In the first step of the algorithm, we automatically choose identifiers since we do not want to require operators to specify a search key. The intuition is that if a variable meets the three criteria in step 1 of Algorithm 1, it is likely to identify objects such as transactions. The frequency/distinct value pattern of identifiers is very different from other variables, so it is easy to discover identifiers[1]. We have very few false selections in all data sets, and the small number of false choices is easy to eliminate by a manual examination. Notice that continuous numerical values, such as file sizes, do not meet these criteria, because it is

[1]Like the state variable case, identifiers are chosen as variables reported at least $0.2N$ times, where N is total number of messages. We also require the variables have at least $0.02N$ distinct values, and reported in at least 5 distinct messages types.

Algorithm 1 Message count vector construction

```
1. Find all message variables reported in the log with the following properties:
   a. Reported many times;
   b. Has many distinct values;
   c. Appears in multiple message types.
2. Group messages by values of the variables chosen above.
3. For each message group, create a message count vector $\mathbf{y} = [y_1, y_2, \ldots, y_n]$,
   where $y_i$ is the number of appearances of messages of type $i$ $(i = 1 \ldots n)$
   in the message group.
```

unlikely for the same value to appear in several different message types. In the HDFS log, the only variable selected in step 1 is block ID, an important identifier.

In the second step, the message group essentially describes an execution path, with two major differences. First, not every processing step is necessarily represented in the console logs. Since the logging points are hand chosen by developers, it is reasonable to assume that logged steps should be important for diagnosis. Second, correct ordering of messages is not guaranteed across multiple nodes, due to unsynchronized clocks across many computers. This ordering might be a problem for diagnosing synchronization-related problems, but it is still useful in identifying many kinds of anomalies.

In the third step, we use the *bag of words* model [29] to represent the message group because: 1) it does not require ordering among terms (message types), and 2) documents with unusual terms are given more weight in document ranking. In our case, the rare log messages are indeed likely to be more important.

We gather all the message count vectors to construct message count matrix $\mathbf{Y}^m$ as an $m \times n$ matrix where each row is a message count vector $\mathbf{y}$, as described in step 3 of Algorithm 1. $\mathbf{Y}^m$ has n columns, corresponding to n message types that reported the identifier (analogous to "terms"). $\mathbf{Y}^m$ has m rows, each of which corresponds to a message group (analogous to "document").

Although the message count matrix $\mathbf{Y}^m$ has completely different semantics from the state ratio matrix $\mathbf{Y}^s$, both can be analyzed using matrix-based anomaly detection tools (see Section 4.2). Table 4.1 summarizes the semantics of the rows and columns of each feature matrix.

4.1.3 Implementing feature creation algorithms

To improve the efficiency of our feature generation algorithms in map-reduce, we tailored the implementation. The step of discovering state variables and/or identifiers (the first steps in Section 4.1.1 and 4.1.2) is a single map-reduce job that calculates the number of distinct values for all variables and determines which variables to include in further feature generation steps. The step of constructing features from variables is another map-reduce job with log parsing as the map stage and message grouping as the reduce stage. For the state ratio vector, we sort messages by time stamp, while for the message count vector, we

Feature	**Rows**	**Columns**
Status ratio matrix $\mathbf{Y}^s$	time window	state value
Message count matrix $\mathbf{Y}^m$	identifier	message type

Table 4.1: Semantics of rows and columns of features

sort by identifier values. Notice that the map stage (parsing step) only needs to output the required data rather than the entire text message, resulting in huge I/O savings during the data shuffling and sorting before reduce. Feature creation time is negligible compared to parsing time, as Section 3.5 described.

4.2 Anomaly Detection

After the feature creation step, we convert a single "interleaved" log into many independent sequences of related messages, and convert these sequences into a numerical representation. This transformation enables us to apply different machine learning algorithms. Here we only focus on anomaly detection algorithms. Anomaly detection is important on console logs. In a production environment, most of the operations in the systems are correctly completed and generate normal log messages, so abnormal messages (or more importantly, abnormal message sequences) best represent problems and thus important to discover.

There are many anomaly detection techniques to choose from. Given the feature matrices we construct, outlier detection methods can be applied to detect anomalies contained in the logs. Notice that since the matrix structure of both message count vector and state ratio vector features are identical, the same algorithm can be applied to both features.

We have investigated a variety of methods, including one-class Support Vector Machine (SVM) [43] and mixture models [43], and have found that Principal Component Analysis (PCA) [23, 58] combined with term-weighting techniques from information retrieval [78, 83] yields excellent anomaly detection results on both feature matrices with little parameter tuning.

4.2.1 PCA-based anomaly detection

Principal Component Analysis (PCA) is a statistical method that captures patterns in high-dimensional data by automatically choosing a set of coordinates—the *principal components*—that reflect covariation among the original coordinates. We use PCA to separate out repeating patterns in feature vectors, thereby making abnormal message patterns easier to detect. The time complexity of PCA is linear in the number of feature vectors; therefore, detection can scale to large logs.

We apply PCA to the feature matrix $\mathbf{Y}^2$, treating each row $\mathbf{y}$ as a point in $\mathbb{R}^n$. The set of n principal components, $\{\mathbf{v}_i\}_{i=1}^n$, are defined by

[2]To calculate the principal components, we assume that $\mathbf{Y}$ is normalized by subtracting the column mean.

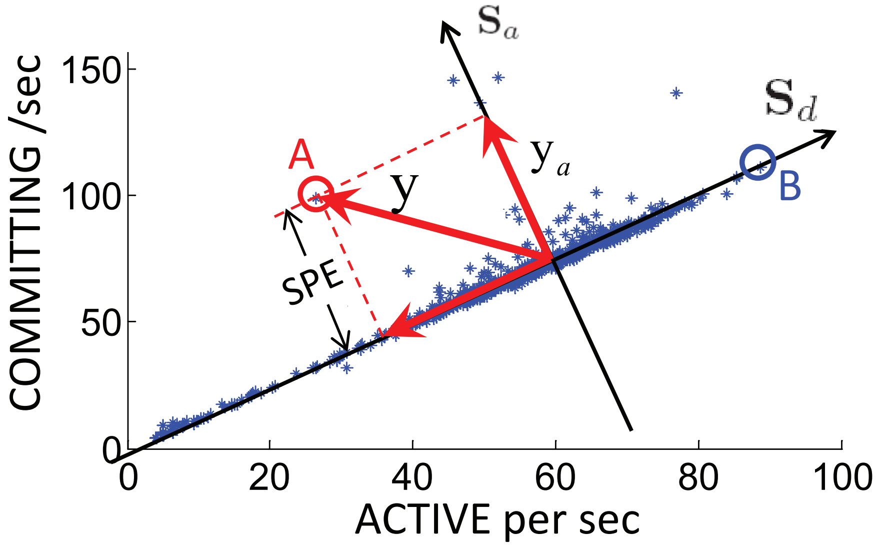


Figure 4.1: The intuition behind PCA detection with simplified data. We plot only two dimensions from the Darkstar state variable feature in the *original coordinates*. It is easy to see high correlation between these two dimensions. PCA chooses the new coordinates $\mathbf{S}_d$ and $\mathbf{S}_a$, which reflect covariation among the original coordinates. PCA determines the dominant normal pattern, separates it out, and makes it easier to identify anomalies.

$$\mathbf{v}_i = \arg \max_{\|\mathbf{x}\|=1} \|(\mathbf{Y} - \sum_{j=1}^{i-1} \mathbf{Y}\mathbf{v}_j\mathbf{v}_j^T)\mathbf{x}\|.$$

In fact, the $\mathbf{v}_i$'s are the the n eigenvectors of the estimated covariance matrix

$$\mathbf{A} := \frac{1}{m}\mathbf{Y}^T\mathbf{Y}$$

and each $\|\mathbf{Y}\mathbf{v}_i\|^2$ is proportional to the variance of the data measured along $\mathbf{v}_i$.

Intuition behind PCA anomaly detection.

By construction, dimensions in our feature vectors are highly correlated, due to the strong correlation among log messages within a group. We aim to identify abnormal vectors that deviate from such correlation patterns. Figure 4.1 illustrates a simplified example using two dimensions (number of `ACTIVE` and `COMMITTING` per second) from Darkstar state ratio vectors. We see most data points reside close to a straight line (a one-dimensional subspace). In this case, we say the data have *low effective dimensionality*. The axis $\mathbf{S}_d$ captures the strong correlations between the two dimensions. Intuitively, a data point far from $\mathbf{S}_d$ (such as point A) shows unusual correlation, and thus represents an anomaly. In contrast, point B, although far from most other points, resides close to $\mathbf{S}_d$, and is thus normal. In fact,

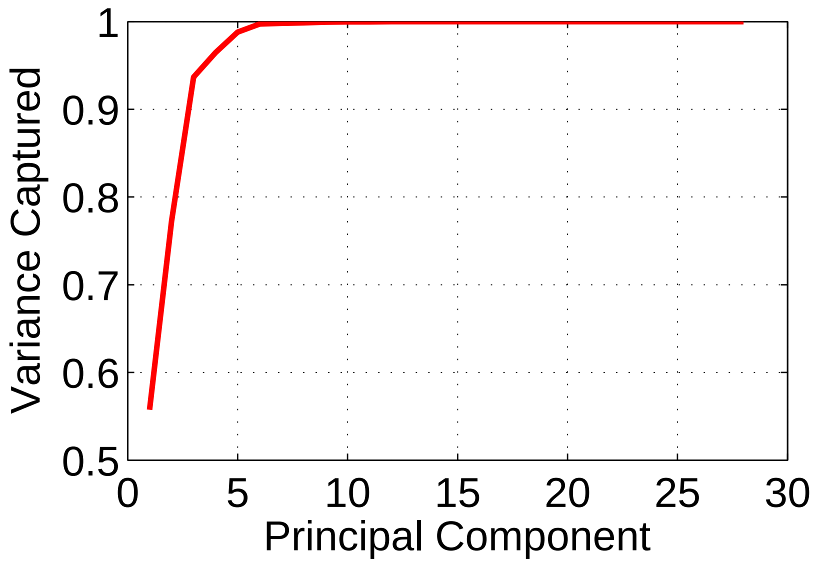


Figure 4.2: Fractional of total variance captured by each principal component.

Feature data sets	n	k
Darkstar - message count	18	3
Darkstar - state ratio	6	1
HDFS - message count	28	4
HDFS - state ratio	202	2

Table 4.2: Low effective dimensionality of feature data. $n =$ Dimensionality of feature vector $\mathbf{y}$; $k =$ Dimensionality required to capture 95% of variance in the data. In all of our data, we have $n \gg k$, exhibiting low effective dimensionality.

both `ACTIVE` and `COMMITTING` are larger in this case, which simply indicates that the system is busier.

Indeed, we do observe low effective dimensionality in the feature matrices $\mathbf{Y}^s$ and $\mathbf{Y}^m$ in many systems. Figure 4.2 shows that in HDFS, most of the variance is captured by a small number of principal components, even if the original vector has almost 30 dimensions. Table 4.2 shows k, the number of dimensions required to capture 95% of the variance in data[3]. Intuitively, in the case of the state ratio vector, when the system is in a stable state, the ratios among different state variable values are roughly constant. For the message count vector, as each dimension corresponds to a certain stage in the program and the stages are determined by the program logic, the messages in a group are correlated. The correlations among messages, determined by the normal program execution, result in highly correlated dimensions for both features.

[3]The choice of using 95% is a common heuristic for determining k in PCA detectors [51]; we use this number in all of our experiments.

In summary, PCA captures dominant patterns in data to construct a (low) k-dimensional *normal* subspace $\mathbf{S}_d$ in the original n-dimensional space. The remaining $(n-k)$ dimensions form the abnormal subspace $\mathbf{S}_a$. By projecting the vector $\mathbf{y}$ onto $\mathbf{S}_a$ (separating out its component on $\mathbf{S}_d$), it is much easier to identify abnormal vectors. This forms the basis for anomaly detection [23, 58].

Detecting anomalies.

Intuitively, we use the "distance" from the endpoint of a vector $\mathbf{y}$ to the normal subspace $\mathbf{S}_d$ to determine whether $\mathbf{y}$ is abnormal. This intuition can be formalized by computing the *squared prediction error* $\mathbf{SPE} \equiv \|\mathbf{y}_a\|^2$ (the squared length of vector $\mathbf{y}_a$), where $\mathbf{y}_a$ is the projection of $\mathbf{y}$ onto the abnormal subspace $\mathbf{S}_a$, and can be computed as

$$\mathbf{y}_a = (\mathbf{I} - \mathbf{P}\mathbf{P}^T)\mathbf{y} \tag{4.1}$$

where

$$\mathbf{P} = [\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k] \tag{4.2}$$

is formed by the first k principal components chosen by PCA algorithm.

As Figure 4.1 shows, abnormal vectors are typically far away from the normal subspace $\mathbf{S}_d$. Thus, the "detection rule" is simple: we mark $\mathbf{y}$ as abnormal if

$$\mathbf{SPE} = \|\mathbf{y}_a\|^2 > Q_\alpha, \tag{4.3}$$

where Q_α denotes the threshold statistic for the $\mathbf{SPE}$ residual function at the $(1-\alpha)$ confidence level.

Automatically determining the detection threshold.

To compute Q_α we make use of the Q-statistic, a well-known test statistic for the $\mathbf{SPE}$ residual function [48]. The computed threshold Q_α guarantees that the false alarm probability is no more than α under the assumption that data matrix $\mathbf{Y}$ has a multivariate Gaussian distribution. However, as pointed out by Jensen and Solomon [48], and as verified in our empirical work, the Q-statistic is robust even when the underlying distribution of the data differs substantially from Gaussian.

The choice of the confidence parameter α for anomaly detection has been studied in previous work [58], and we follow standard practice in choosing $\alpha = 0.001$ in our experiments. We found that our detection results are not sensitive to this parameter choice.

4.2.2 Improving PCA detection results

Applying TF/IDF.

Our message count vector is constructed in a way similar to the bag-of-words model, so it is natural to consider term weighting techniques from information retrieval. We applied *Term Frequency / Inverse Document Frequency* (TF/IDF), a well-established heuristic in information retrieval [78, 83], to pre-process the data. Instead of applying PCA directly to the feature matrix $\mathbf{Y}^m$, we replace each entry $y_{i,j}$ in $\mathbf{Y}^m$ with a weighted entry

$$w_{i,j} \equiv y_{i,j} \log(n/df_j) \tag{4.4}$$

where df_j is total number of message groups that contain the j-th message type. Intuitively, multiplying the original count with the IDF reduces the weight of common message types that appear in most groups, which are less likely to indicate problems. We found this step to be essential for improving detection accuracy.

TF-IDF does not apply to the state ratio feature. This is because the state ratio matrix is a dense matrix that is not amenable to interpretation as a bag-of-words model. However, applying the PCA method directly to $\mathbf{Y}^s$ gives good results on the state ratio feature.

Using better similarity metrics and data normalization.

We also mine our data with a more powerful variation of PCA – kernel PCA [84], which allows us to specify a desired similarity metric (i.e. kernel function) for feature vectors. We experiment with a variety of kernel functions, and get the best results with the cosine kernel with

$$\mathcal{K}(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{\sqrt{\mathbf{x} \cdot \mathbf{x}}\sqrt{\mathbf{y} \cdot \mathbf{y}}}, \tag{4.5}$$

which can be interpreted as a similarity metric between vector $\mathbf{x}$ and $\mathbf{y}$. In fact, cosine similarity is widely used in information retrieval to analyze document matrices. Cosine similarity is not affected by the total number of terms (log messages) in a particular document (message group). The document length problem is especially significant in our case because some hot blocks are accessed many more times than rare ones, thus generating many more log messages.

With cosine kernel, kernel PCA is (mathematically) equivalent to linear PCA when each message count vector $\mathbf{y}$ is normalized by its length

$$\|\mathbf{y}\| = \sqrt{\mathbf{y} \cdot \mathbf{y}} \tag{4.6}$$

Thus in practice we can avoid the computation and memory complexity to compute the kernel matrix while still achieving the good results of kernel PCA.

Both improvements apply to our case because the message count vector is constructed by analogy to the bag-of-words model, thus having many common properties as the vector representation of regular documents. Therefore, many information retrieval techniques (e.g. clustering/classification) might also apply to this feature. We discuss the accuracy improvement of using TF/IDF and the cosine distance in Section 4.3.2.

Before presenting our visualization step, we first summarize the anomaly detection results on both cases using both message count vector and state ratio vector features. We believe the understanding of anomaly detection results makes the discussion on decision tree visualization (in Section 4.4) more concrete.

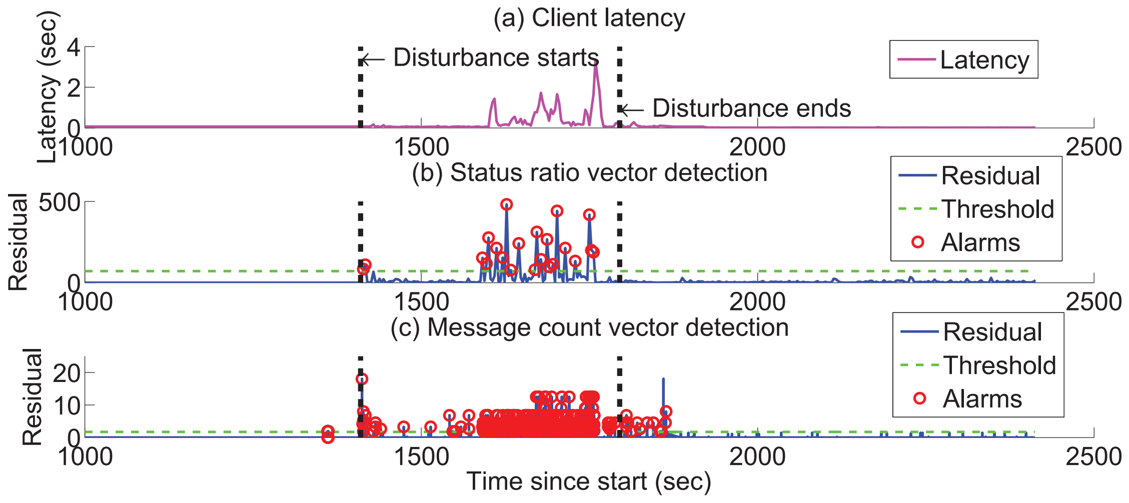


Figure 4.3: Darkstar state ratio vector detection results. (a) shows that the disturbance injection caused a huge increase in client response time. (b) shows PCA anomaly detection results on the state ratio vector created from message variable `state`. The dashed line shows the threshold Q_α. The solid line with spikes is the SPE calculated according to Eq. (4.3). The circles denote the anomalous vectors detected by our method, whose SPE values exceed threshold Q_α. (c) shows detection results with the message count vector. The SPE value of each vector (the solid line) is plotted at the time when the last message of the group occurs.

4.3 Evaluation and Visualization

We evaluated our log mining techniques using two real-world systems. We discussed log parsing accuracy and scalability in Section 3.5, and here we focus on the effectiveness of feature extraction and anomaly detection.

We began our experiments of problem detection with Darkstar, in which both features give simple yet insightful results (Section 4.3.1). Satisfied with these results, we applied our techniques to the much more complex HDFS logs. We also achieve high detection accuracy (Section 4.3.2).

As we will see in this section, although PCA anomaly detection provides highly accurate results, these results are less intuitive to system operators and developers. In Section 4.4 we discuss our decision tree visualization method, which summarizes the PCA detection results in a single, intuitive picture that is more operator-friendly because the tree resembles the rule-based event processing systems operators tend to use [42].

4.3.1 Darkstar experiment results

As mentioned in Section 2.3, we observed high variability in performance (i.e., client side response time) when deploying the Darkstar server in a cloud-computing environment such as Amazon's Elastic Computing Cloud (EC2) [3] during performance disturbances, especially for CPU contention. We wanted to see if we could understand the reason for this high performance variability solely from console logs. Indeed, we were unfamiliar with Darkstar, so our setting was realistic as the operator often knows little about system internals.

In the experiment, we deployed an unmodified Darkstar 0.95 distribution on a single node (because the Darkstar version we use supports only one node). Darkstar does not log much by default, so we turned on debug-level logging. We deployed a simple game, DarkMud, provided by the Darkstar team, and created a workload generator that emulated 60 users in the DarkMud virtual world performing random operations such as flipping switches and picking up and dropping items. The client emulator recorded the latency of each operation. We ran the experiment for 4800 seconds and injected a performance disturbance by capping the CPU available to Darkstar to 50% of the normal level during time 1400 to 1800 seconds.

Detection by state ratio vectors.

The only state variable chosen by our feature generation algorithm is `state`, which is reported in $456,996$ messages (about 28% of all log messages in our data set). It has 8 distinct values, including `PREPARING`, `ACTIVE`, `COMMITTING`, `ABORTING` and so on, so our state ratio matrix $\mathbf{Y}^s$ has 8 columns (dimensions). The time window (automatically determined according to Section 4.1.1) is 3 seconds; we restricted the choice to whole seconds.

Figures 4.3 (a) and (b) show the results between time 1000 and 2500, where plot (a) displays the average latency reported by the client emulator, which acts as a ground truth for evaluating our method, and plot (b) displays the PCA anomaly detection results on the state ratio matrix $\mathbf{Y}^s$. We see that anomalies detected by our method during the time interval (1400, 1800) match the high client-side latency very well; i.e., the anomalies detected in the state ratio matrix correlate very well with the increases in client latency. Comparing the abnormal vectors to the normal vectors, we see that the ratio between number of `ABORTING` to `COMMITTING` increases from about 1:2000 to about 1:2, indicating that a disproportionate number of `ABORTING` transactions are related to the poor client latency.

Generally, the abnormal state ratio may be the cause, symptom, or consequence of the performance degradation. In the Darkstar case, the ratio reflects the cause of the problem: when the system performance gets worse, Darkstar does not adjust transaction timeout accordingly, causing many normal transactions to be aborted and restarted, resulting in further load increase to the system.

Notice that a traditional *grep*-based method does not help in this case for two reasons:

1. For a Darkstar operator without much knowledge about its internals, the transaction states are obscure implementation details. Thus, it is difficult for an operator to choose the most useful state variable values from many variables. In contrast, we systematically discover and analyze all state variables.

2. `ABORTING` happens even during normal operations, due to the optimistic concurrency model used in Darkstar, where aborting is used to handle access conflicts. It is not a single `ABORTING` message, but the ratio of `ABORTING` to other values of the `state` variable that captures the problem.

Detection by message count vectors.

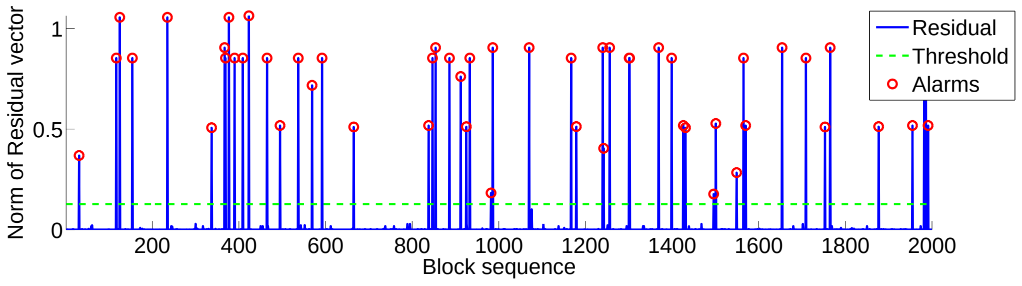


Figure 4.4: PCA detection on message count vector in Hadoop dataset with residual component $\mathbf{y}_a$, the projection on the abnormal subspace. The figure plots a random selection of 2000 blocks out of the entire data set of $575,139$ blocks. The dashed line shows the threshold Q_α. The solid line with spikes is the SPE calculated according to Eq. (4.3). The circles denote the anomalous message count vectors detected by our method, whose SPE values exceed threshold Q_α.

From Darkstar logs, Algorithm 1 automatically chooses two identifier variables, the *transaction id* and the *asynchronous channel id*. Figure 4.3(c) shows detection results on the message count vector constructed from the transaction id variable. There are 68,029 transaction ids reported in 18 different message types. Thus, the dimension of matrix $\mathbf{Y}^m$ is 68,029 $\times$ 18. By construction, each message count vector represents a set of operations (message types) occurring when executing a transaction. PCA identifies the normal vectors corresponding to a common set of operations (simplified for presentation): {`create`, `join txn`, `commit`, `prepareAndCommit`}. Abnormal transactions can deviate from this set by missing a few message types, or having rare message types such as `abort txn` instead of `commit` and `join txn`. We detected 504 of these as abnormal. To validate our result, we augmented each feature vector using the timestamp of the last message in that group, and we found that almost all abnormal transactions occur when the disturbance is injected. We see that the anomalies continue to appear (with a smaller frequency) for a short time period after the disturbance stopped, due to queueing effects as the system recovered from the disturbance. Notice that the state ratio vector method did not mark the recovery period as abnormal, demonstrating that the message count vector method was more sensitive because it modeled individual operations while state ratio vector method captured only aggregate behavior.

There were no anomalies on the `channelID` variable during the entire experiment, suggesting that the `channelID` variable is not related to this performance anomaly.

This result is consistent with the state ratio vector detection result. In console logs, it is common that there are several different pieces of information that describe the same system behavior. This commonality suggests an important direction for future research: exploiting multi-source learning algorithms, which combine multiple detection results to further improve accuracy.

4.3.2 Hadoop experiment results

Compared to Darkstar, HDFS is larger scale and its logic is much more complex. In this experiment, we show that we can automatically discover many abnormal behaviors in HDFS. We generated the HDFS logs by setting up a Hadoop cluster on 203 EC2 nodes and running sample Hadoop map-reduce jobs for 48 hours, generating and processing over 200 TB of random data. We collected over 24 million lines of logs from HDFS.

Detection on message count vector.

From the HDFS logs, Algorithm 1 automatically chooses one identifier variable, the `blockid`, which is reported in 11,197,954 messages (about 50% of all messages) in 29 message types. Also, there are 575,139 distinct `blockid`s reported in the log, so the message count matrix $\mathbf{Y}^m$ has a dimension of 575,139 $\times$ 29. Figure 4.4 shows that the PCA detector gives very good separation between normal and abnormal row vectors in the matrix: Using an automatically determined threshold (Q_α in Eq. (4.3) in Section 4.2), it can successfully detect abnormal vectors corresponding to blocks that went through abnormal execution paths.

To further validate our results, we manually labeled each distinct message vector, not only marking them as normal or abnormal, but also determining the type of problems for each vector. The labeling was done by carefully studying HDFS code and by consulting with local Hadoop experts. We show in the next section that the decision tree visualization helps both ourselves and Hadoop developers to understand our results. *We emphasize that this labeling step is done only to validate our method—it is not a required step when using our technique.* Labeling half a million vectors is possible because many of the vectors are exactly the same. In fact, there are only 680 distinct vectors, confirming our intuition that most blocks go through a common execution path.

Table 4.3 shows the manual labels and detection results. We see that the PCA detector can detect a large fraction of anomalies in the data, and significant improvement can be achieved when we preprocess data with TF/IDF, confirming our expectations from Section 4.2.

Throughout the experiment, we experienced no catastrophic failures; thus, most problems listed in Table 4.3 only affect performance.

The first anomaly in Table 4.3 uncovered a bug that has been hidden in HDFS for a long time. In a certain (relatively rare) code path, when a block is deleted (due to temporary over-replication), the record on the namenode is not updated until the next write to the block, causing the file system to assume the presence of a replica that no longer exists, which causes subsequent block deletion to fail. Hadoop developers have recently confirmed this bug. This anomaly is hard to find because there is no single error message indicating the problem. However, we discover it because we analyze abnormal execution paths.

We also notice that we avoid a problem that causes confusion in traditional grep based log analysis. In HDFS datanode logs, we see many messages of the form `#:Got Exception while serving # to #:#`. According to Apache issue tracking (jira) HADOOP-3678, this is a normal behavior of HDFS: the HDFS data node generates the exception when a HDFS client does not finish reading an entire block before it stops. These exception messages have

#	**Anomaly Description**	**Actual**	**Raw**	**TF-IDF**
1	Namenode not updated after deleting block	4297	475	4297
2	Write exception client give up	3225	3225	3225
3	Write failed at beginning	2950	2950	2950
4	Replica immediately deleted	2809	2803	2788
5	Received block that does not belong to any file	1240	20	1228
6	Redundant addStoredBlock	953	33	953
7	Delete a block that no longer exists on data node	724	18	650
8	Empty packet for block	476	476	476
9	Receive block exception	89	89	89
10	Replication Monitor timedout	45	37	45
11	Other anomalies	108	91	107
	Total	**16916**	**10217**	**16808**

#	**False Positive Description**	**Raw**	**TF-IDF**
1	Normal background migration	1399	1397
2	Multiple replica (for task / job desc files)	372	349
3	Unknown Reason	26	0
	Total	**1797**	**1746**

Table 4.3: Detected anomalies and false positives using PCA on Hadoop message count vector feature. Actual is the number of anomalies labeled manually. Raw is PCA detection result on raw data, TF-IDF is detection result on data preprocessed with TF-IDF and normalized by vector length (Section 4.2).

confused many operators, as indicated by multiple discussion threads on the Hadoop user mailing list. While traditional keyword matching (e.g., searching for words like *Exception* or *Error*) would have flagged these as errors, our message count method successfully avoids this false positive because this happens too many times to be abnormal.

Our algorithm does report some false positives, which are inevitable in any unsupervised learning algorithm. For example, the second false positive in Table 4.3 occurs because a few blocks are replicated 10 times instead of 3 times for the majority of blocks. These message groups look suspicious, but Hadoop experts told us that these are normal situations when the map-reduce system is distributing job configuration files to all the nodes. It is indeed a *rare* situation compared to the data accesses, but is *normal* by the system design. Eliminating this type of "rare but normal" false positive requires domain expert knowledge. An interesting future direction would be to investigate semi-supervised learning techniques that can take operator feedback and further improve our results.

Detection on state ratio vectors.

The only state variable chosen in HDFS logs by our feature generation algorithm is the *node name*. Node name might not sound like a state variable, but as the set of nodes (203 total) are relatively fixed in HDFS, their names meet the criterion of state variable described in Section 4.1.1. Thus, the state ratio vector feature reduces to per node activity count, a feature well-studied in existing work [45, 61]. As in this previous work, we are able to detect transient workload imbalance, as well as node reboot events. However, our approach is less ad-hoc because the state ratio feature is chosen automatically based on information in the console log, instead of manually specified.

4.4 Visualizing Detection Results with Decision Trees

From the point of view of an operator, the transformation underlying PCA is a *black box* algorithm: it provides no intuitive explanation of the detection results and cannot be interrogated. Human operators need to manually examine anomalies to understand the root cause, and PCA itself provides little help in this regard. In this section, we show how to augment PCA-based detection with decision trees to make the results more easily understandable and actionable by operators. The decision tree result resembles the (manually written) rules used in many system-event-processing programs [42], so it is easier for non-machine learning experts to interpret. This technique is especially useful for features with many dimensions, such as the message count vector feature in HDFS.

Decision trees have been widely used for classification. Because decision tree construction works in the original coordinates of the input data, its classification decisions tend to be easy to visualize and understand [98]. Constructing a decision tree requires a training set with class labels. We use the automatically generated PCA detection results (normal vs. abnormal) as class labels, in contrast to the normal use of decision trees. Our decision tree is constructed to explain the underlying logic of the detection algorithm, rather than the nature of the dataset.

Figure 4.5 is the decision tree generated using RapidMiner [72] from the anomaly detection results of the HDFS log. It clearly shows the most important message types. For example, the first level shows that if `blockMap` (the data structure that keeps block locations) is updated more than 3 times, which is abnormal. This indicates the over-replication problem (Anomaly 4 or False Positive 1 in Table 4.3). The second level shows that if a block is received 2 times or less, the block not correctly written; this anomaly indicates under-replication or block-write failure (Anomaly 2 and 3 in Table 4.3). Level 3 of the decision tree is related to the bug we discussed in Section 4.3.2.

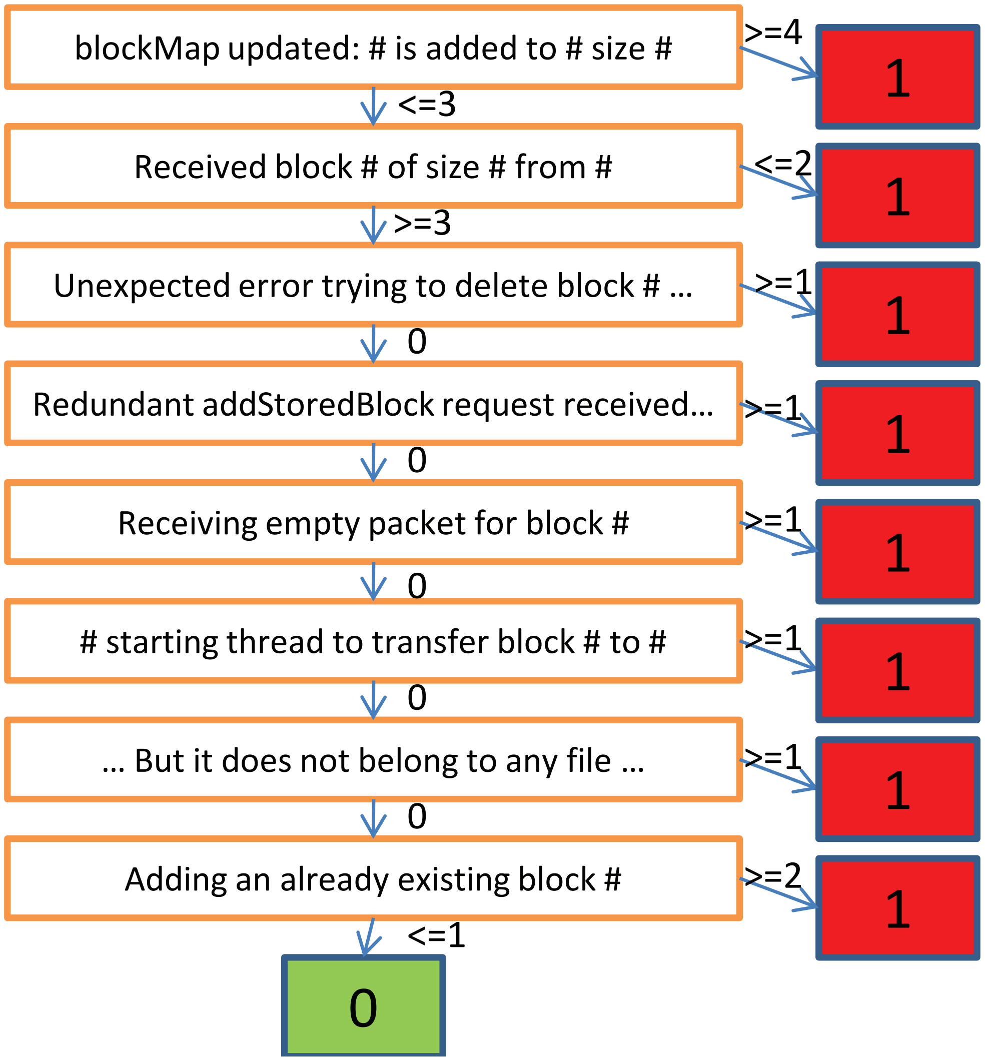
Figure 4.6 shows the result of applying the same decision tree visualization technique to the Darkstar dataset. It is short and simple because the data from Darkstar is simpler (i.e. fewer dimensions and fewer anomaly types). However, the simple decision tree does provide the correct explanation of anomalous cases, `read incomplete` when a transaction is aborted.

In summary, the visualization of results with decision trees helps operators and developers notice *types* of abnormal behaviors instead of individual abnormal events, which can greatly improve the efficiency of finding root causes and preventing future alarms.

4.5 Summary

In this chapter, we introduced the last three steps of in our four-step log analysis methodology: feature creation, anomaly detection, and visualization. Although we use two case studies to make the introduction concrete, our methodology does not use any application-specific information and thus can be generalized to many different systems. Unlike existing work, our feature creation techniques take advantage of the accurate message parsing, and use the message variables to help group logs to capture execution sequences (message count vector) or snapshot of system state (state ratio vector). This grouping process results in high-quality features that allow efficient machine learning algorithms such as PCA to yield accurate detection results. We used decision tree visualization to provide intuitive explanations to system operators.

However, there is an unrealistic assumption used throughout this chapter that we have logs for the entire time period of program execution. In real production systems, execution can last weeks or even months, which makes the offline techniques less useful for operator who wants to discover problems quickly. In the next chapter, we discuss how we can perform near-real-time problem detection. We will show that the state ratio vector feature is easy to use in an online setting, as it segment logs by time, but we need to refine the message count vector creation process to make it applicable in an online setting.



blockMap updated: # is added to # size #
>=4
1
<=3
Received block # of size # from #
<=2
1
>=3
Unexpected error trying to delete block # …
>=1
1
0
Redundant addStoredBlock request received…
>=1
1
0
Receiving empty packet for block #
>=1
1
0
# starting thread to transfer block # to #
>=1
1
0
… But it does not belong to any file …
>=1
1
0
Adding an already existing block #
>=2
1
<=1
0

Figure 4.5: The decision tree visualization of Hadoop message count vectors. Each node is the message type string (# is the place holder for variables). The number on the edge is the threshold of message count, generated by the decision tree algorithm. Small boxes contain the labels from PCA, with number 1 for abnormal and number 0 for normal.

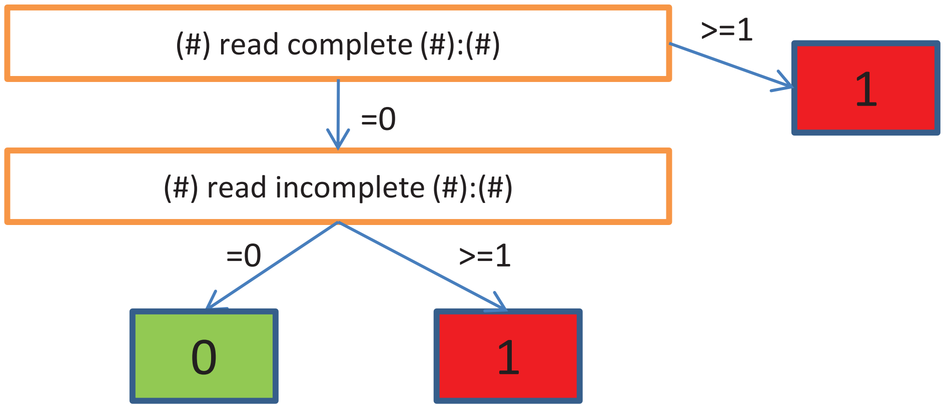


Figure 4.6: The decision tree visualization for Darkstar message count vector detection results. Small boxes contain the labels from PCA, with number 1 for abnormal and number 0 for normal.

Chapter 5

Problem Detection in Online Log Streams

Compared to offline approaches discussed in Chapter 4, the fundamental problem of online analysis is that we cannot see the complete event trace at once. The anomaly detection techniques discussed in the previous chapter assume we have the knowledge of the entire execution trace and thus are only useful in an offline setting. Although the detection provides insightful information on potential bugs and other performance problems, it remains important to continuously monitor a production system and detect problems in near real-time. Quick detection of problems not only allows system operators to reduce expensive downtime, but also reduce the amount of storage space required to monitor data.

Whether we can trivially adapt an offline detection method to an online detection setting depends on the feature vector construction, and in particular on whether we can create the feature without seeing the entire log. For example, the state ratio vector can be constructed from events within a limited time window, so it works in an online setting without changing.

On the other hand, the message count vector is based on multiple overlapping sequences of messages. In the previous chapter, we assumed *post-mortem* analysis on complete traces. In a continuously generated log stream, it is not obvious when we can declare that an execution sequence on a particular identifier has completed, thus it is nontrivial to construct the feature vector.

In this chapter, we discuss our novel two-stage *online* log processing approach that combines frequent pattern mining with PCA anomaly detection for runtime problem detection. In particular, we show how to trade off time-to-detection vs. accuracy in the online setting by augmenting frequent-sequence information with timestamp information. Like offline detection, online detection is completely automatic.

We evaluated our technique on the same labeled Hadoop dataset described in the previous chapter. In nearly all respects, we match or exceed the detection accuracy of the offline approach with small detection latencies.

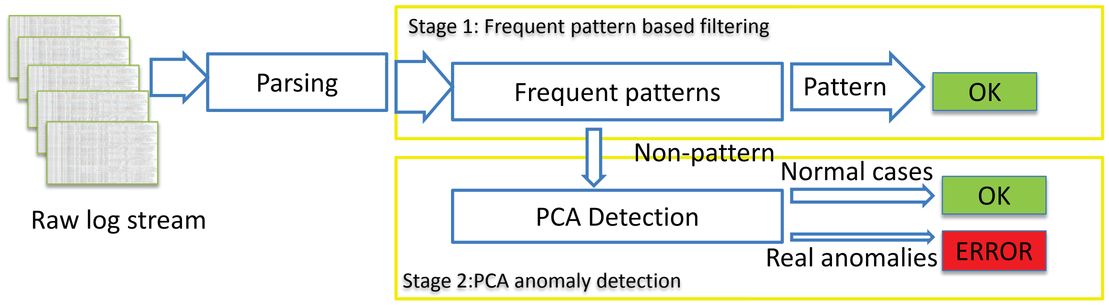


Figure 5.1: Overview of the two stage online detection systems.

5.1 Two-Stage Online Anomaly Detection

5.1.1 Challenges in online detection

The fundamental problem of online analysis is that we cannot see all log messages at once. A straightforward solution is to segment the log stream into time windows, and perform detection at the end of each time window. The correct time window strikes a balance between accuracy and timeliness in the detection. At one extreme, if we wait to see the entire trace before attempting any detection, our results should be as accurate as offline detection but with excessive time to detection. At the other extreme, if we try to make a determination of anomalous behavior as soon as a single event appears, we lose the ability to perform anomaly detection based on sequences of messages, which is the key to achieve high accuracy. We emphasize that detection time is determined only by how long the algorithm has to wait before making a decision. The computation time of the detection algorithm is negligible compared to this wait.

How to choose the time windows depends on the nature of the feature, but the time window should allow most of the related messages to be contained in a single time window so that the features can capture the correlations.

It is trivial for some features. For example, the state ratio vector discussed in the previous chapter is easy to construct and use in online detection, as it is naturally defined by time windows and captures aggregated behavior. To perform the same PCA detection as discussed in the last chapter, we simply keep a running count for a single state ratio vector and send the vector to PCA detection on the expiration of each time window. The detection is fast: we only need to apply Equation 4.3 to compute **SPE** and compare it to the threshold Q_α. We can compute the transformation matrix $(\mathbf{I}-\mathbf{P}\mathbf{P}^T)$ and the threshold Q_α used in PCA detection (defined in Section 4.2) from a short history and update the model periodically from most recent history.

In contrast, message count vectors are not easy to construct as it depends on traces of individual messages. In offline detection, a trace may be marked as abnormal because an event is missing; if a write operation to a file fails, its trace may lack a “closing” message. In online analysis, there is no way to know, other than waiting until the end of the run, if the missing event will ever come, yet the whole point of online detection is to make an assessment in a timely manner. If we fail to put the "closing" message and the previous messages in the same window, the detection algorithm will believe the sequence is not correctly completed, resulting false positives. Notice that in this case, we need to segment *every sequence* correctly, in contrast to the state ratio case, where we only need to make sure the *aggregate* behavior is preserved in a window.

As we will discuss later in this chapter, there can be large variations in the durations of each sequence. In other words, each sequence requires its own "time window". In our experiments, and we suffer over 80% false positives when we choose a fixed time window on HDFS data set.

Even worse, due to the current mechanisms in which console logs are generated and collected, there are some console-log-specific problems. The most important one is message reordering due to unsynchronized clocks. In a distributed system, such as HDFS, a single sequence might involve multiple nodes. The clocks among different nodes are not always synchronized. Thus it is not always possible to have the exact ordering among messages from different nodes. Previous work suggested ways to preserve causal ordering for tracing [28]. However, all these approaches require extra bookkeeping and communications among nodes, defeating our purpose of leveraging the simplicity of built-in console logs.

We designed a two stage detection method, which uses frequent patterns to determine the detection time for each individual sequence. We tolerated reordering problem using frequent sequences, which effectively eliminated noise due to reordering. We also model the tail distribution of sequence durations in order to know the correct time window for each sequence.

5.1.2 Our solution: two-stage detection

We make this tradeoff by designing a two-stage detection method. The first stage uses frequent pattern mining to capture the most common (i.e., normal) session, that is, those traces with a high support level. The patterns include both frequent-event set and time information. This information can be used to determine when a trace is "probably complete" and ready to be made available for anomaly detection. The second stage considers only non-pattern events that make it through the first stage, applying PCA-based anomaly detection to them. In each stage, we build a model based on archived history and update it periodically with new data, and use it for online detection. Both model estimation and online detection involve domain-specific considerations about console logs.

Figure 5.2 shows clearly why a two-stage approach is needed. The histogram of the 50 most frequent event traces in our data shows that some traces clearly occur extremely frequently while others are extremely rare. It is reasonable to mark the dominant traces as "normal" behavior and the rare outliers as "anomalous", but this leaves a large middle ground of traces that are neither obviously dominant nor obviously anomalous. These traces in the middle ground are sometimes normal ones with added random noise such as interleavings. We want our detection method to tolerate the random noise. If we reduce the minimal support level to include more of these middle-ground cases, random noise (e.g.

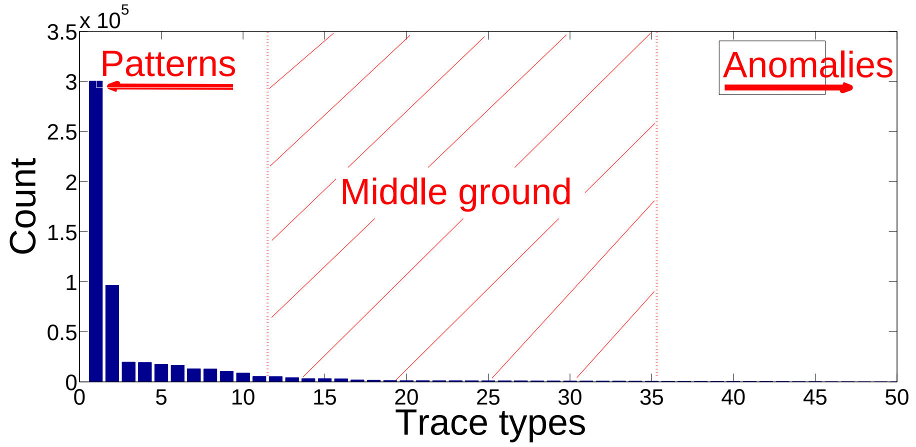


Figure 5.2: Histogram of 50 most frequent traces. Some traces are extremely frequent, and some are extremely rare, but there is a large "middle ground" which is neither pattern nor anomaly for sure.

overlapping or incorrect ordering) will be introduced into the patterns, reducing the quality of the patterns.

Instead, we pass the middle-ground cases to a PCA-based anomaly detector as non-pattern events. Since PCA is a statistical method that is able to match "inexact" patterns, it is more robust to random noise than the frequent-pattern mining used in stage 1 and can detect rare events among the middle-ground cases. Intuitively, the pattern-based method provides timely detection for the majority of events, minimizing the time to wait for the complete trace, while subsequent PCA-based detection handles the false alarms generated by the first stage and greatly improves detection accuracy. An additional benefit to the two-stage approach is that the frequent patterns from stage 1 can help operators to better understand the behavior of their systems and tune the detection to include domain-specific knowledge.

Although PCA is more robust to random noise than pattern mining and thus a suitable method for dealing with the noisy middle-ground events, frequent pattern mining has the advantage of capturing time information among events and providing an intuitive representation of dominant patterns. Our two-stage approach integrates the advantages of both methods. We now describe each stage in detail in the following two sections.

More advanced methods, such as n-gram, Hidden Markov Models (HMMs), and probabilistic context free grammars, were applied to model the ordered sequences [99, 17]. They do not work well in the console log case due to unsynchronized timestamps. These methods are also vulnerable to overfitting with workload dependent data. We discuss more about these methods in Chapter 7.

5.2 Stage 1: Frequent Pattern Mining

Frequent pattern mining is well studied by data mining researchers, and many efficient algorithms have been proposed [1, 41, 88, 102, 79]. We further discuss these algorithms in Section 7.3. There are two major challenges with console logs that are not usually addressed

```
1     00:00:00 allocating blk_1
2     00:00:05 receiving blk_1
3     00:00:06 receiving blk_1
4     00:00:20 received blk_1
5     00:00:36 received blk_1
6     00:00:36 registering blk_1 on name node
7     01:22:34 reading blk_1
8     04:22:34 reading blk_1
9     20:20:21 start deleting blk_1
10    20:20:25 deleted blk_1
```

Figure 5.3: Sample sessions. Although the log segment looks like a single event stream, it represents multiple independent sessions on `blk_1`.

in existing work: 1) Frequent pattern mining algorithms usually assume that the data are divided into separate "transactions". In contrast, log is a continuous message sequence and the boundaries between different "transactions" are not clear. 2) Existing sequenced-based pattern mining algorithms assume exact ordering in data, however, in our case, we only observe partially-ordered log messages.

Following conventions in online system management work [45, 67, 46], we use the term *event* to refer to a parsed log message. Specifically, we define an *event* to be a tuple consisting of a timestamp, the event type (the message type in the previous chapter), and a list of message variables. In an online setting, the streaming console log becomes an *event stream* after the parsing step. In the previous chapter, we focused on properties of the entire trace. For the online detection we need to analyze part of the traces. Thus, we further define a *session* to be a subset of closely-related events in the same event trace that has a predictable duration. The *duration* of a session is the time difference between the earliest and latest timestamps of events in the session. For example, Figure 5.3 shows a simplified segment of HDFS log showing events happening on `blk_1`. From a human operator's perspective, the first six lines represent a session that `writes` `blk_1`. Line 7 is a `read` session on the block (notice that this session only generates a single event in the log). Line 8 is a separate `read` session, and the last two lines are from the same `delete` session. Logically, these sessions are independent from each other, and each session has a different duration. For example, the duration of the write session is the time difference between the first event and the sixth event, which is 36 seconds. The main goal of the frequent pattern mining step is to automatically discover these sessions, and model the session durations without using semantics of the log messages.

We define a *frequent pattern* to be a session and its duration distribution such that:

1. The session occurs frequently in many event traces;

2. Most (e.g., 99.95th percentile) of the session's duration is less than T_{max}, a user-specified maximum allowable *detection latency*. The detection latency is the time

between an event occurring and the decision of whether the event is normal or abnormal.

Condition (1) guarantees that the pattern covers common cases so it is likely to be a normal behavior. Condition (2) guarantees that the pattern can be detected in a bounded time. We mine the archived data periodically for frequent patterns. These patterns are used to filter out normal events in the online phase.

We cannot apply generic frequent sequence mining techniques for two reasons. First, sessions may interleave in the event traces (e.g. two reads happen at the same time) thus "transaction" boundaries are not clear. We need to simultaneously segment an event trace into sessions and mine patterns. However, because the durations of sessions can have large variations, fixed time windows will not give satisfactory segmentation, which suggest that we shall model the distribution of durations. Second, events can be reordered in the traces because of unsynchronized clocks in a distributed system, which precludes the use of techniques requiring total ordering of events. In our algorithm described below, we use frequent patterns to tolerate the poor time-based segmentation accuracy resulting from random session interleavings. The frequent patterns, once discovered, can be used to de-interleave the events to estimate a clean duration model.

5.2.1 Mining frequent event patterns

Our novel approach combines time and event sequence information for accurate pattern detection using a 3-step iterative method. In a nutshell, we first use time information to inaccurately segment an event trace into sessions and then mine these inaccurate segments to identify the most frequent pattern. We then go back to the original data and find out the actual time distribution of the sessions of the most frequent pattern. Finally, we remove all events that match this frequent pattern from original data and iterate on the remaining data to find the next most frequent pattern.

1. Use time gaps to find first session coarsely in each execution trace. In this step, for each execution trace, we first scan through each event until we find an event followed by a time gap more than 10 times the duration since the start of the execution sequence[1]. We treat all events preceding the gap as a session and represent these events as message count vectors (MCVs). This segmentation can be very inaccurate; due to interleaving sessions, irrelevant events might be included in the session and due to the randomness in session duration, events may be missing from the session. The inaccuracy is tolerated by the next step when finding most frequent patterns.

2. Identify the dominant session. We want to find a pattern that contains all events in a session. This requirement is satisfied in most cases due to the way sessions get segmented: with high probability, if not always, events that happen close together in time often represent a single logical operation, especially when the support level is high (recall the definition of sessions at the beginning of this section).

We use two criteria to select the dominant pattern.

[1]The time gap size is a configurable parameter.

(1) We start with the *medoid* of all sessions, which are represented by MCVs. The medoid is defined as the minimal aggregated distance from all other data points, which indicates that it is a good representative of all data points. Intuitively, a medoid is similar to the centroid or mean in the space, except that the medoid must be an actual data point. Criterion 1 guarantees that the selected dominant session is a *good representative* of the sessions examined.

(2) We require the session to have a minimal support. If the medoid does not meet this minimal support, we choose the next closest data point that does. Criterion 2 guarantees that the selected session is in fact dominant, in addition to being a good representative. Currently we choose to set the minimal support to $0.2M$ from all M event traces. The selection criteria are robust over a wide range of minimal support values because the normal traces are indeed in the majority in the log. In fact, in our experiments, various support values between $0.1M$ and $0.5M$ all resulted in the same selection results.

3. Refine result using the frequent session and compute duration statistics. Notice that the pattern from step 2 is based on coarsely segmented sessions, and may not reflect the correct duration distribution of all sessions of that type. Because we know the events we are expecting to complete a session, we can go back to the original data and find all events that match the frequent session and then estimate the duration distribution from the matching sessions (detailed in Section 5.2.2). Using the duration distribution, we can compute a *cutoff time* T_{cut} representing the time that most sessions of the pattern "should" complete, for the pattern as the η^{th} percentile of the distribution. We show in Section 5.4.2 that this step significantly improved detection results. We also remove all matching events from the original traces, preparing the data for the next iteration. Notice that T_{cut} can be very long, due to large dispersion in durations in some operations. In the case that $T_{cut} > T_{max}$, the pattern is discarded and not used in the detection stage.

We then return to step 1 and iterate until no patterns with the minimal support level remain. Since step 3 always removes something from the dataset, the iteration is guaranteed to terminate. The remaining events are used to construct the PCA model.

The dominant patterns are expected to be stable. However, in order to accommodate changes in the operation environment, we update patterns used in the detector as a periodic and infrequent offline process; that is, the detector uses the patterns discovered but never updates them online. In this way, we can both keep the online detector simple and avoid poisoning the patterns with transient abnormal periods.

5.2.2 Estimating distributions of session durations

To enable timely online detection, we need to know how long any given pattern "should" take to complete. To this end, we estimate the distribution of session durations for each pattern. Based on this distribution, we compute the cutoff time T_{cut} as the 99.95^{th} percentile of the distribution for each pattern, after which most sessions of this pattern would complete.

To choose a distribution to fit our data, we observe that within each pattern, the histogram of session durations has both dominant values and fat tails. We use the first two patterns in in Table 5.1 of Section 5.4.1) as examples. Figures 5.4 (a) and (b) show the

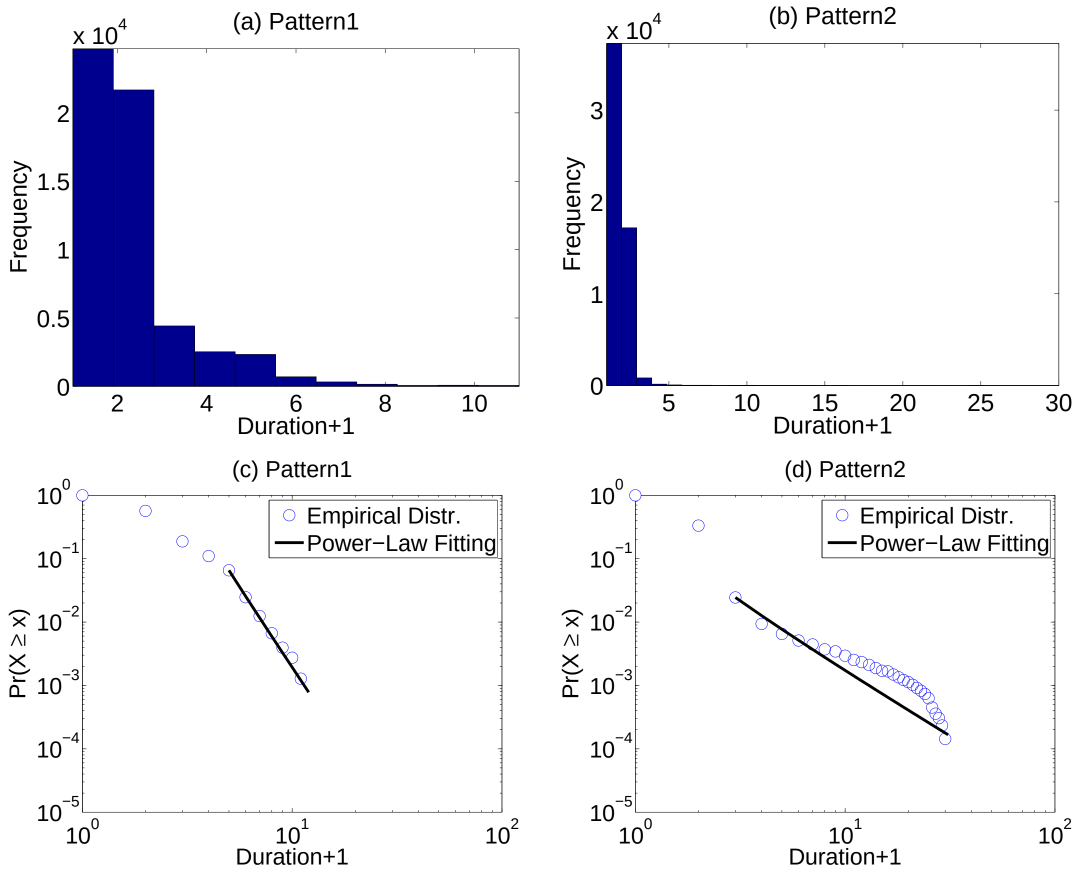


Figure 5.4: Tail of durations follow power-law distribution.

duration distribution of these two patterns. The power-law distribution are widely used to model data with long tails for its unique mathematical properties [27, 65]. We choose it to model our data, and a log-log plot confirms that the tails of our data approximately follow the power-law distribution (Figures 5.4 (c) and (d)).

To estimate the parameters of the distribution, we adopt the approach proposed in [18], which combines maximum-likelihood fitting methods with goodness-of-fit tests based on the Kolmogorov-Smirnov (KS) statistics [14]. In real applications, few datasets obey power-laws for all values. More often, the power-law applies only to values greater than some minimum $x_{min} > 0$, i.e. to the tail of the distribution. For samples below this threshold, we use the histogram as its empirical distribution. So we essentially use a mixture distribution with two components to model the duration values: a power-law distribution for the tail (values above x_{min}), which has weight w, and a histogram for values below x_{min}, which has weight $(1 - w)$.

For durations that take only integer values, we consider the case with a probability distribution of the form $p(x) = Pr(X = x) = Cx^{-\beta}$. It is not difficult to show that the normalizing constant is given by:

$$C(\beta, x_{min}) = \left(\sum_{i=0}^{\infty} (i + x_{min})^{-\beta} \right)^{-1}. \tag{5.1}$$

Assuming x_{min} is known (the way to estimate x_{min} is discussed later), the Maximum Likelihood Estimator (MLE) of the scaling parameter β is approximately

$$\hat{\beta} \approx 1 + n \left[\sum_{i=1}^{n} \ln \frac{x_i}{x_{min} - 0.5} \right]^{-1}, \tag{5.2}$$

where $x_i, i = 1 \ldots n$ are the observed duration values that $x_i \geq x_{min}$.

To estimate x_{min}, we choose a value that makes the probability distributions of the measured data and the best-fit power-law model as similar as possible above x_{min}. We use KS statistics to measure the distance between two distributions, and estimate $\hat{x}_{min}$ as the value of x_{min} that minimizes the KS statistics between the empirical CDF of the data for the observations with value at least x_{min} and the fitted power-law model that best fits the data in the region with all $x_i \geq x_{min}$.

Figure 5.4 (c) and (d) show the empirical distributions (circles) and the fitted power-law models (solid lines) for patterns 1 and 2, respectively. With the model, the CDF $P_p(x) = Pr(X < x)$ of the power-law distribution is

$$P_p(x) = 1.0 - C(\beta, x)/C(\beta, x_{min}), \tag{5.3}$$

where $C(\beta, x)$ is defined in Eq. (5.1). Then, for $\eta \geq 1.0 - w$, the η^{th} percentile of the mixture distribution is the value of x_η that satisfies the following equation:

$$P_p(x_\eta) = (\eta - (1.0 - w))/w, \tag{5.4}$$

where $P_p(x)$ is defined in Eq. (5.3). We show the estimated 99.90^{th}, 99.95^{th} and 99.99^{th} percentiles of the mixture distributions of Patterns 1 and 2 and the improvements to the detection precision in Section 5.4.

5.2.3 Implementation of Stage 1

The pattern-based detector receives the event stream from the log parser. If an event is part of some execution traces we are monitoring because it contains an identifier, the detector groups it with other events with the same identifier and checks if any subset of the event group matches a frequent pattern. If a subset matches, all matching events are removed from the detector's memory, although there might still be some non-matching events left in the queue. Removing matched events keeps the size of in-memory event history small and greatly improves the efficiency of the detector.

Logically, we try matching all event sets to all patterns. We used a naïve method that attempts each one. We believe the naïve method is good enough in many systems because the number of patterns is usually small and the traces are short because developers only log the most important stages on the execution path. However, in cases where many long patterns are used, we can use more advanced data structures such as suffix trees [81] to improve the matching efficiency.

If we do not find any matching pattern, the event is added to the queue with a timeout number T_o based on the event timestamp T. If the event matches one or more patterns, we choose the one with the largest cutoff time (T_{cut}) and set $T_o = T + T_{cut}$; if the event does not match any pattern (because the event is not frequent enough to be included in any pattern), we set $T_o = T + T_{max}$. Notice that because T_{cut} is usually much smaller than T_{max}, we can achieve fast detection on the majority of events.

The detector periodically checks all traces. Currently, the period is set to 1 second—this parameter has a small effect on detection time, but no effect on accuracy. When it finds events that have reached their timeout, it constructs their message count vectors, as described in Section 4.1, and sends them to the second stage PCA-based detector.

The intuition behind this approach is that an event is passed through to the PCA-based detector as soon as we can be reasonably sure that it does not "belong to" any of the frequent patterns being monitored. We call these *non-pattern events*.

5.3 Stage 2: PCA Detection

The vectors representing the non-pattern events emitted from Stage 1 are significantly noisier than the frequent patterns. The noise comes from uncaptured interleaving, high variations in duration and the true anomalies. To uncover the true anomalies from this noisy data, we use the same PCA detector, which is shown to be accurate in offline problem detection in last chapter.

The PCA detector works essentially the same way as the offline case in Section 4.2. The model used in PCA, the transformation matrix and the threshold, can be updated periodically. Note that because of the noisier data in this phase and the workload-dependent nature of the non-pattern data, the model update period for PCA is usually shorter than that for frequent pattern mining.

We want to emphasize that although the structures of vectors and the detection are similar to the offline case, the semantic meanings are different. In the offline case, the vector represents a complete processing sequence that an identifier went through. In the online case, each vector represents a collection of *uncommon* or non-pattern operations on an identifier *within a small time window*.

5.4 Evaluation

To compare our online approach directly against the offline algorithm proposed in Section 4.3, we replayed the same set of logs, containing over 24 million lines of log messages with an uncompressed size of 2.4GB. Recall that the log contains 575,319 event traces, corresponding to 575,319 distinct HDFS file blocks. We also re-used the manual labels from

No.	**Frequent sessions**	**Duration in sec (%ile)**			**Events**
		99.90	**99.95**	**99.99**	
1	Allocated block, begin write	11	13	20	20.3%
2	Done write, update block map	7	8	14	44.6%
3	Delete block	-	-	-	12.5%
4	Serving block	—			3.8%
5	Read Exception (see text)	—			3.2%
6	Verify block	—			1.1%
	Total				**85.6%**

Table 5.1: Frequent patterns discovered from Hadoop logs. Pattern 3's duration cannot be estimated because the durations are too small to capture in training set. Patterns 4–6 consist of only a single event each and thus have no durations.

the experiments described in the last chapter as ground truth for evaluating our results. Notice that the labeling process does not take into account the durations of any traces. We show the effects of this omission later in this section.

To mimic how a system operator would use our technique, we evaluate our method with the following 2-step approach. First, we randomly sample 10% of the execution traces, on which we construct the detection model, including the frequent patterns, the distributions of pattern durations, and the PCA detector. Then we replay the entire trace and perform online problem detection using the derived model. This whole procedure is unsupervised, since we use the labels only for evaluation and not for building the model. We varied the subset of sampled data for building the model many times, and got identical detection results. The result is robust against random sampling mainly because the patterns we identify are frequent enough in the data set.

There are two parameters that we need to set. The *maximum detection latency* T_{max} (defined in Section 5.2) was set to 60 seconds, meaning the operator wants to be notified of a suspected anomaly at most 60 seconds after the suspect event trace appears in the log. The PCA threshold parameter α, described in Section 5.3, was set to 0.001, meaning that we are accepting fewer than 0.1% of all data points as abnormal under assumptions described in Section 4.2 and [48]. These baseline values are likely choices with no understanding of the data, but in Section 5.4.2 we show that our detection results are insensitive to these parameters over a wide range of values.

5.4.1 Stage 1 pattern mining results

Recall that the goal of Stage 1 is to remove frequent patterns that presumably correspond to normal application behavior. Table 5.1 summarizes the frequent patterns found in the test data using our baseline parameter value $T_{max} = 60$ seconds. Note first that the patterns identified encompass 85.6% of all events in the trace, so at most 14.4% of all events must be considered by Stage 2 (PCA anomaly detection).

The table shows, for example, that pattern 1 is the sequence of events corresponding to "Allocate a block for writing". 20.3% of the events in the trace are classified as belonging to an instance of this pattern and so will be filtered out and *not* passed to Stage 2. The duration of this pattern has a distribution whose 99.9, 99.95, and 99.99^{th} percentiles are 11, 13 and 20 seconds, respectively. We choose these high percentile values because we want most normal sessions to complete within these intervals. Notice that even the 99.99th percentile of the pattern durations is significantly smaller than T_{max}. The short pattern durations are important since detection latency is based on the these durations or T_{max}, whichever is less. To be succinct, we only present detection results with T_{cut} set to the 99.95^{th} percentile values for each pattern. Results with other values are similar.

Patterns 1 and 2 are both related to writing a file block. They logically belong to the same operation, but a write session can be arbitrarily long; the application that writes the file may wait an arbitrary amount of time after the "begin write" before actually sending data. Since we are trying to keep detection latency below a certain threshold, we separate the beginning and ending sessions into two different patterns for timely detection. Obviously, there are certain limitations related to this separation, which we discuss in detail in Section 5.5.

Patterns 4 to 6 contain only individual events. These events were used to report some numbers and do not contribute to event trace based detection, so a single event completes the operation. For example, reads, deletions and block verifications are all single-event patterns.

Pattern 5 consists of an event that reports an exception, but as we discussed in Section 4.4, this is indeed *normal* operation and the message text represents a bad logging practice that has confused many users. Just as the offline detection case, because we use pattern frequencies for detection, we easily recognize these exception messages as normal operation.

In addition to providing a shortcut for normal operations in our detection, the patterns themselves are interesting also because they can provide valuable insight to operators and help them understand the "normal" behavior of their system.

5.4.2 Detection precision and recall

Recall that a session is a subset of a trace. Since our technique is based on sessions, we determine a trace is abnormal if and only if it contains at least one abnormal session, allowing direct comparison using the original labels. We use the standard metrics of precision and recall to evaluate our approach. Let TP, FP, FN be the number of true positives, false positives, and false negatives, respectively. We have

$$\text{Precision} = \text{TP}/(\text{TP}+\text{FP})$$

and

$$\text{Recall} = \text{TP}/(\text{TP}+\text{FN})$$

(a) Varying α while holding $T_{max} = 60$

α	**TP**	**FP**	**FN**	**Precision**	**Recall**
0.0001	16,916	2,444	0	87.38%	100.00%
0.001	**16,916**	**2,748**	**0**	**86.03%**	**100.00%**
0.005	16,916	2,914	0	85.31%	100.00%
0.01	16,916	2,914	0	85.31%	100.00%

(b) Varying T_{max} while holding $\alpha = 0.001$

T_{max}	**TP**	**FP**	**FN**	**Precision**	**Recall**
15	2,870	129	14,046	95.70%	16.97%
30	16,916	2,748	0	86.03%	100.00%
60	**16,916**	**2,748**	**0**	**86.03%**	**100.00%**
120	16,916	2,748	0	86.03%	100.00%
240	14,233	2,232	2,683	86.44%	84.14%

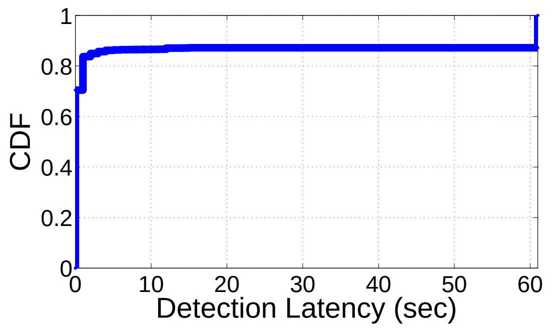
Table 5.2: Hadoop online detection precision and recall.

100% recall means that no actual problems were missed; 100% precision means there are no false alarms among those events identified as problems. Recall that in our data set, there are 575,319 event traces of which 16,916 are labeled as anomalies.

Table 5.2(a) varies the PCA confidence level α to show its effect on our precision and recall results, while Table 5.2(b) varies the maximum detection delay T_{max}. The boldface rows of each table represent the baseline values $\alpha = 0.001$ and $T_{max} = 60$. The results show 100% recall over a wide range of values of α and T_{max}, meaning the algorithm captures every anomaly in the manual labels. The precision and recall are largely insensitive to the choice of α, consistent with the observations in both the offline case and [58]. The good recall is mainly due to strong patterns in the data: the event traces are direct representations of the program execution logic, which is likely to be deterministic and regular, as reflected by log printing statements. The strong patterns allows better tolerance to random noise, especially in the frequent pattern mining stage, where we can use a high support requirement to filter out random interleavings and reorderings.

The precision is not perfect due to false positives and some ambiguous cases. We review the false positives in detail when we compare with offline results in Section 5.4.4.

Table 5.2(b) shows that precision and recall are insensitive to the maximum detection latency T_{max} over a certain range, but setting it outside this range (first and last rows of Table 5.2(b)) adversely affects recall or precision. The intuition is that when T_{max} is too small, many logical sessions, especially those not covered by the dominant patterns, are cut off randomly, and when T_{max} is too large, many unrelated sessions are combined into the same message count vector, introducing too much noise for the PCA detector. Either effect degrades precision and recall.



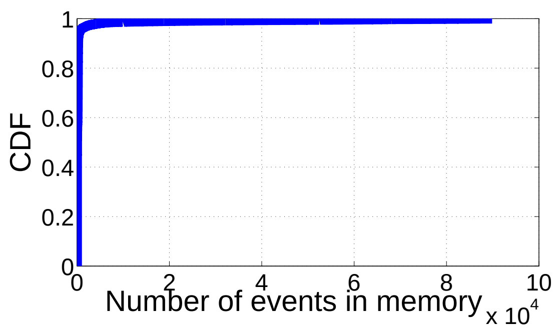


Figure 5.5: Detection latency and number of events kept in detector's buffer

As we described in Section 5.2.2, we used a fairly sophisticated model to estimate the duration of sessions. If we had instead assumed a simple Gaussian distribution, the 99.95th percentile of T_{cut} would be estimated as 5.3 seconds for Pattern 1 and 4.0 seconds for Pattern 2 in Table 5.1—less than half as long as the T_{cut} estimated by our distribution-fitting. Using the Gaussian-derived cutoff time, the number of false alarms increases by 45%, and precision falls to 80% from 86%. Therefore the small added complexity for duration distribution estimation in Section 5.2.2 results in much better recall and precision.

5.4.3 Detection latency

Detection latency, defined in Section 5.2, captures timeliness of detection, a key goal of our online approach. Recall that the difficulty of minimizing detection latency arises from the fact that it is not always possible to mark a trace "abnormal" until a specific event or set of events occurs. For example, the "allocate block" message in Pattern 1 of Table 5.1 simply indicates the start of a sequence of operations; the detector has to buffer the event and wait for further events. The final decision for this message is not reached until the last event of Pattern 1; that is, the last of the three expected "receiving" messages. Then the *detection time* for the trace containing this "allocate block" event is the time elapsed from the "allocate block" event being emitted to the time the detection result is made.

Figure 5.5 on the left shows the cumulative distribution function (CDF) of detection times over all events. As expected, over 80% of the events can be determined as normal or abnormal within a couple of seconds. The short detection latency is because we use the cutoff time T_{cut} to stop waiting for more events instead of the max latency T_{max} for most events. Events that do not match any pattern require the maximum allowed detection latency: T_o defaults to $(T + T_{max})$. By definition, these events are rare, so the overall impact of their longer detection time is limited.

Figure 5.5 on the right shows the CDF of the number of events buffered in detector at every second. Because the detection time is low, most events are processed and removed from the buffer quickly. Thus, as expected, the typical number of events in the buffer is small.

No.	Anomaly Description	Actual	Offline	Online
1	Namenode not updated after deleting block	4297	4297	4297
2	Write exception client give up	3225	3225	3225
3	Write failed at beginning	2950	2950	2950
4	Replica immediately deleted	2809	2788	2809
5	Received block that does not belong to any file	1240	1228	1240
6	Redundant addStoredBlock	953	953	953
7	Delete a block that no longer exists on data node	724	650	724
8	Empty packet for block	476	476	476
9	Receive block exception	89	89	89
10	Replication monitor timedout	45	45	45
11	Other anomalies	108	107	108
	Total	**16916**	**16808**	**16916**

#	False Positive Description	Offline	Online
1	Normal background migration	1397	1403
2	Multiple replica (for task / job desc files)	349	368
	Total	**1746**	**1771**

#	Ambiguous Case	Offline	Online
	(see Section 5.4.4)	**0**	**977**

Table 5.3: Detection accuracy comparison with offline detection results. Actual is the number of anomalies labeled manually. For fair comparison, we reused labels from Table 4.3 in Section 4.3. Offline is PCA detection result presented in Section 4.3 and online is our result using our two stage detection method in an online setting, with the baseline parameters.

5.4.4 Comparison to offline results

Table 5.3 compares the offline detection results from the last chapter to our online detection results using baseline parameter values $\alpha = 0.001$ and $T_{max} = 60$. The error labels in the first column of the table were obtained directly from Table 4.3. The online method has an even higher recall than the offline method. The reason is that for online detection, we segment an event trace into several sessions based on time duration, and base the detection on individual sessions rather than whole traces. Thus, the data sent to the detector is free of noise resulting from application-dependent interleaving of multiple independent sessions (e.g., some blocks are read more often than others).

The two types of false positives in Table 5.3 are both “rare but normal events”. For example, false positive #2 (over-replicating) is due to a special application request rather than a system problem. These are indeed rare events, as there are only 368 occurrences across all traces. These cases are hard to handle with a fully unsupervised detector. As

Table 5.4: Add caption

Experiments	Precision	Recall
Online with ambiguous cases	86%	100%
Online excluding ambiguous cases	89%	100%
Offline	91%	99.3%

Table 5.5: Precision and recall comparison between online and offline detection results. The offline result is from Chapter 4, while the two online results represent counting ambiguous cases as false positive or not. See text in Section 5.4.4 for details.

we discussed in the previous chapter, in order to handle these cases, we allow operators to manually add patterns to encode domain-specific knowledge. Note that the patterns are much easier to write than PCA models. This is a beneficial side effect of the two-stage method over the offline method.

Table 5.3 lists ambiguous cases arising from the unclear definition of "anomaly". For example, our online algorithm marks some write sessions abnormal because one of the data nodes takes far longer to respond than all others do, resulting an unusually long writing session[2]. From the system administration point of view, these cases probably should be marked as anomalies, because although these blocks are eventually correctly written, this scenario effectively slows down the entire system to the speed of the slowest-responding node. Table 5.5 summarizes the precision and recall comparing to the offline detection algorithm.

In the offline experiments (Section 4.3), however, an event trace is labeled as normal if it contains all the events of a given pattern, without regard to *when* the events occur. Since they do not consider time information such as the durations of sessions, a scenario in which one data node takes a long time (but eventually responds) is no different from a scenario in which all nodes respond in about the same amount of time. We consider session durations because we need to do so in order to bound the time to detection, but here we see that this additional information potentially improves the value of the online approach for operators in another way as well—by labeling as anomalous those event traces that are "correct but slow." If we consider slow operations problematic, at least some of these false alarms would instead be counted as a new type of anomaly not detected by the offline approach.

To determine how many of the 977 ambiguous cases fall into this category, we would have to examine all event traces manually to evaluate duration lengths, in contrast to examining only distinct traces without considering time information as was done in the offline case discussed in the last chapter. However, we did an informal evaluation to estimate the number of cases that are probably due to this problem. We forced T_{cut} to 600 seconds for all patterns, which forces the detector to wait a long time for any incomplete patterns. Under these circumstances, the detection results approximate the results achieved by the offline detection algorithm when ignoring time information: the number of ambiguous cases drops to 314, which suggests at least 2/3 of these types of false alarms are fair to count as real anomalies. Nonetheless, to keep a fair comparison with the offline result, we stick to the

[2]The write durations of these ambiguous cases ranges from 13 seconds to hundreds of seconds, while the median duration for all sessions of this type is less than 1 second.

original labels in all our evaluations. Table 5.5 compares the precision and recall in either case when these ambiguous cases are counted as false positives vs. when they are not.

5.5 Discussion

5.5.1 Limitations of online detection

An obvious limitation of online detection is that we cannot capture correlations across events over very long time periods. For example, as discussed in Section 5.4.1, there is a large and unpredictable time gap between Patterns 1 and 2 in Table 5.1, so we must separate them into two patterns. However, a consequence of this separation is that we lose the ability to observe correlations between events in Pattern 1 and "matching" events in Pattern 2, which would potentially allow us to capture a new category of operational problems. For example, events in Pattern 1 indicate how many data nodes begin a write; each such node should have a corresponding "end write" event in Pattern 2.

It is an inherent limitation that online detection cannot capture patterns with long durations, because of the detection latency requirement. This problem could be solved by remembering a longer history, which may be stored in a more compact/aggregated form to reduce overhead, though that complicates the design of the detector. Thus we propose a different approach: by leveraging relatively cheap computing cycles, we can perform offline detection periodically on archived data to find anomalies violating such uncaptured constraints.

5.5.2 Use cases

In addition to showing individual anomaly alarms, our technique lets operators link each alarm back to the original logs and even the related source code segments, using the parsing and visualization techniques described in Section 4.4. In addition, since we detect performance anomalies quickly, operators have more time to prevent them from causing more serious errors. Anomalies due to deterministic bugs can recur frequently even over short timescales, as occurs with Anomaly 1 in Table 5.3, which is due to a deterministic bug in the Hadoop source code. Since setting off alarms on each occurrence would overwhelm the operator's attention, we cluster the anomalies hierarchically and report the count of each anomaly *type* and a count of the occurrences of each type.

5.6 Summary

In this chapter, we show how to use a two-stage data mining technique to identify and filter out common, normal operational patterns from free-text console logs, and then perform PCA-based anomaly detection on the remaining patterns to identify operational problems within minutes of their occurrence as represented by information in the console logs. Our evaluation shows that the two stage detection method can achieve or even exceed the detection accuracy of the offline algorithm described in the previous chapter.

In the next chapter, we will discuss a large scale case study of applying these methods on console logs from production systems from Google. Similar to the previous datasets, the Google data contains both identifiers and state variables so our methods still apply. However, the scale and complexity of such data brings unique challenges. We focus on the observation of these datasets and our approaches to deal with these challenges in the next chapter.

Chapter 6

Real world application

Although both case studies in the previous two chapters, Hadoop and Darkstar, are real open source systems, the logs we used came from experimental deployments. In this chapter, we discuss our experience applying our log analysis methodology to Google's production logs. The logs come from a distributed storage system consisting of thousands of nodes in a Google production cluster, which we call GX in this chapter. There are several independent monitoring systems built into GX, and the console log is one of them. GX console logs are collected as plain text files on local disks on each node. Our data set contains *most of* the log messages in a two-month period[1].

We show that we can easily adapt our console log analysis framework to process these logs, and find interesting information about the system. Most of the methods we used are exactly the same as previous chapters. However, there are some new challenges in this data set:

Data size. The data set is five orders of magnitudes larger than the Hadoop data set. In contrast to Hadoop, which logs each write/read operation, GX only logs significant events, such as errors and periodic background tasks. Even omitting per-operation logs, a single node can generate over half a million messages per day, and thus thousands of nodes in the system generate over a billion messages per day. Our entire data set contains tens of billions of messages, with a size about 400 GB uncompressed. We have to parallelize the parsing and feature creation steps onto thousands of nodes so that we can process all the logs within a couple of hours.

Large variety of message types and message variables. Because of the complexity of GX and its extensive interactions with other infrastructure components, there are hundreds of different message types in the logs. In fact, our source code analysis shows that there are over 20,000 different possible message types in GX, compared to only about 1000 in Hadoop. Most of these types never appear in the log because of the logging level settings. There is a wide variety of message variables (both identifiers and state variables) as well. Though we could apply our methods on each identifier and

[1]As the logs are collected on local disks only, there are several corruptions due to disk/node failures or software errors.

each state variable, it would cost more computationally than we could afford. Thus, we heuristically choose the message variables that are likely to indicate problems.

Longer time period with multiple versions. Unlike other data sets we discussed before, we have almost two months worth of data. Due to the longer time period, we need to use the sequence segmentation techniques discussed in Chapter 5 for online processing, even if we perform an offline analysis.

Two limitations prevent us from formally evaluating the analysis result. First, console logs in GX are not regularly used by system operators, and thus there are no manual labels explaining the logs. With an unclear "ground truth", it is not possible to evaluate metrics such as true/false positives. Second, due to confidentiality issues, all human readable string constants are removed (details in Section 6.1), rendering it impossible to understand the semantics of patterns and anomalies discovered. Due to these limitations, we only evaluate our results qualitatively and show that they are still potentially useful to the developers and operators of system GX.

6.1 The art of log sanitization

GX is written in C++, and our C++ parser can achieve highly accurate parsing results on the data set. We discussed the parsing performance in Section 3.5.

In this section, we highlight how parsing helps remove sensitive data from the free text logs while preserving enough information for our analysis. We call the process *log sanitization*. Sanitizing console logs is not scientific as we cannot quantify the effectiveness of sanitization, nor is it a major goal of this research. However, it is a necessary step to obtain the data for research and an interesting application of our log parsing technique.

There are potentially two separate types of sensitive information in console logs: 1) program logic and 2) sensitive data, such as internal entity names. Due to their free text nature, console logs may reveal this sensitive information in any message. Hashing each English word does not work because many identifiers are hierarchical (e.g. a path or a URL), and the naive hashing would destroy these hierarchies and make the identifier-based grouping impossible.

Our message parsing makes the sanitization easier and more effective. Figure 6.1 illustrates major steps to sanitize a message. The message parser first identifies string constants in the message, and then replaces these constants with a unique message type ID. This step removes semantic meanings of messages, making it hard for adversaries to guess the internal program logic. Note that this step is reversible as long as the message templates are available, making it possible for Google's internal operators to reconstruct the original message.

The next step is to apply a one-way secure hash function to all message variables (or parts of the variables) containing sensitive information. In this step, our goal is to preserve as much information as possible while removing all sensitive data.

Selecting which variables to hash can be difficult. One simple scheme is to obtain a list of "sensitive terms", look for variables containing these terms and hash them. This scheme

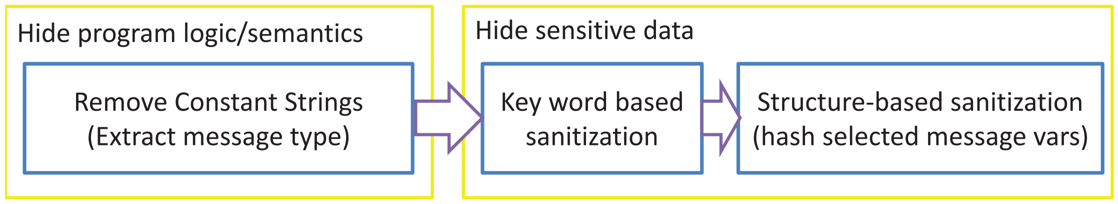

Hide program logic/semantics
Remove Constant Strings (Extract message type)
Hide sensitive data
Key word based sanitization
Structure-based sanitization (hash selected message vars)

Figure 6.1: Log sanitization overview. We need to sanitize two separate types of sensitive information: 1) program logic, and 2) sensitive data. Our sanitization is based on structures of log messages as well as a list of sensitive words.

does not work well enough because 1) it is not possible to obtain an exhaustive list of all sensitive terms, and 2) some of these sensitive terms are phrases instead of single words. Arbitrary string concatenations, especially when constructing hierarchical names, make it hard to identify all phrases from a long string variable. Solving the general problem of string segmentation is a hard problem in information retrieval [56, 100]. Fortunately, we have data type information for each variable from the message parser, so we can specify sanitization rules for each *data type*, rather than each word. For example, if we know many variables are of URL type, we only need to specify a rule to handle URLs, rather than specifying each variable. Although GX contains almost 2000 different variables, there are only about 20 different data types. Using data-type-based rules significantly reduces the manual work in the sanitization process.

There are exceptions to the data-type-based rules. For example, we want to keep some integer-typed variables (e.g. performance counters) while removing other integers revealing machine configurations. We developed a scripting engine that allows us to specify these exceptions.

We applied all sanitization methods discussed above to GX logs. To informally evaluate the effectiveness of such methods, we generated a random sample of 100,000 lines of sanitized logs and manually examined them. We did not find any sensitive information. After the data set was sanitized, we were ready to try our techniques to detect anomalies in the system.

6.2 State-based detection

Most important anomalies are those related to problems affecting a large part or the entire system. We find the state ratio vector feature, introduced in Section 4.1.1, especially suitable in detecting such anomalies.

We used the message type ID obtained from the parsing step as a special state variable. This is a reasonable choice because most of the messages in GX logs are generated by background tasks reorganizing the data storage. The performance and correctness impact of such tasks can be significant. Thus, which background task is running at a given time is an important system state to capture. As each background task always generates a fixed

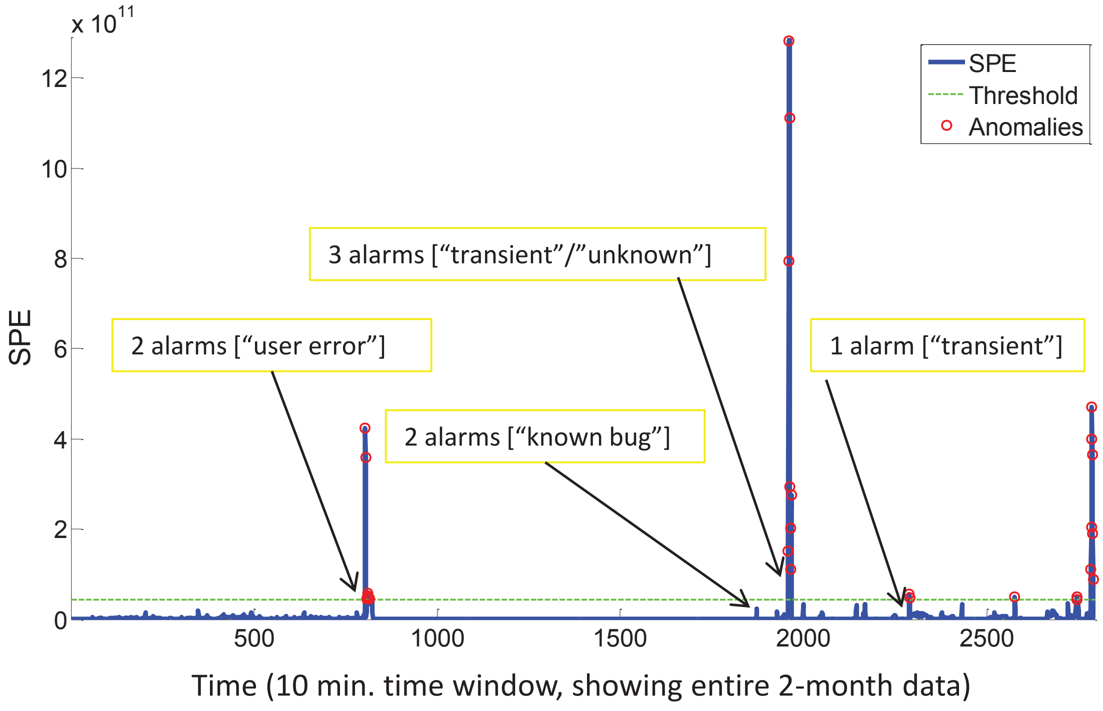


Figure 6.2: GX problem detection results using message type counts on the entire two-month period. Norm is calculated using Equation 4.3 in Chapter 4. Texts with arrows are automatically generated alarms as well as operators' comments on these alarms. The arrows points to the time when the alarms happen.

set of message types that are distinct from those types from other tasks, capturing message type IDs is equivalent to capturing the current state of running background tasks.

We count the number of each different message type occurring in the entire system in every 10-minute time window. There are over 400 different message types, so the resulting state ratio vector has over 400 dimensions.

We then applied PCA detection to these vectors, and Figure 6.2 plots the squared prediction error (SPE, defined in Equation 4.3 in Chapter 4) for each time window. We see that the state ratio vectors stay normal during most of the time windows, indicating that the relative rate for each message type remains constant, matching our assumption that the relative frequency of different background tasks remains constant.

There are a few anomalous time windows, shown as spikes in Figure 6.2. Because we do not have access to semantic information of each message type, nor do we understand the internals of GX, it is not possible to interpret these anomalies. Instead, to verify that these anomalies did affect system performance, we checked performance alarm data. There is a monitoring system that periodically probes GX and raises alarms to operators if there is a significant performance decrease. The operators then investigate these alarms and attach their comments. The text comments with arrows in Figure 6.2 show these alarms with comments. Each arrow points to the time when the alarms are generated. We can easily see that these alarms happen at almost the same time as the anomalies detected by PCA, suggesting changes in state ratio vector feature may be correlated with performance problems. Note that there is a complex logic in the alarm system to suppress unnecessary/duplicate alarms in order to reduce the operator's manual tasks. Due to this fact, the alarms and the anomalies we detected are not likely to form a one-to-one correspondence. Thus, these alarms are different from the manual labels we had in the Hadoop and Darkstar cases, preventing us from accurately evaluating the false positive/negative rates.

With the alarm system built-in, detecting anomalies becomes a less important goal for our log analysis. On the other hand, it is not easy for the operators to determine the cause of the performance problems just by looking at the alarms, as the alarms provide no information other than a long latency. In Figure 6.2, half of the alarms are not clearly marked with the actual cause. Abnormal log patterns, on the other hand, contain information about internal states, and thus can be useful supplements for operators to better diagnose these performance anomalies.

6.3 Sequence-based detection

In GX, storage is partitioned onto multiple nodes. Each partition goes through a similar life cycle: it is first created, then migrates among multiple nodes, and finally is terminated by a re-partition or deletion operation. A partition is manipulated by background tasks, which generate sequences of log messages. In fact, the majority of console log messages are generated by such background tasks.

Each partition is identified by a partition identifier, which is a complex string of arbitrary length and format. Grouping messages by such identifiers, we can capture all messages about events happening on a partition. Each message sequence can last a long time, some even spanning the entire two-month period. In this case, the offline approach, which builds message count vector features from the entire sequence, becomes less useful. We need to segment each sequence into sessions, as we did in our online approach.

Therefore, we directly applied the frequent pattern mining methods discussed in Chapter 5. This pattern mining step results a set of intuitive patterns. For example, when a partition is migrated to a node, the original node normally prints five messages for unloading the partition, while the receiving node prints six messages for loading the node. Our pattern mining technique is able to accurately capture these sequences, as well as the time distribution for each sequence to complete. We discovered nine different patterns, each of which contains two to six events and represents a common background task in the system. There are some less frequent tasks, such as repartitioning, not captured as frequent patterns. Although we could have lowered the minimal support requirement during the pattern mining, we decide to leave these less frequent patterns to the PCA model in the second stage.

The PCA detection stage marked less than 0.1% of over 200 million sequences as anomalies. Most of the anomalies either contain rare message types (probably error messages), or take too long to complete. Without a deep understanding of GX internals, we were not able to determine the performance/correctness impact of each anomaly, nor could we estimate the false positive rate. However, informal communications with GX operators show that these anomalies could help GX developers to understand corner cases better and further improve the performance and reliability of the system.

6.4 Summary

In this chapter, we summarized our experience of applying the console log mining methodology to a production system at Google. Despite the size and complexity of the log, we show that the parsing, feature creation, pattern mining, and anomaly detection techniques are easy to adapt to system GX and yield promising results. Lacking of even basic knowledge about GX prevents us from formally evaluating our results, but informal communications with operators show that the results can be very useful. Global-state-based detections help find causes for performance problems, while sequence-based-detection is useful for finding hard-to-notice corner error cases, helping developers further improve the system.

GX collects other monitoring information besides console logs. These monitoring data are collected from different layers in the software/hardware stack. Much work has been done analyzing monitoring data of different granularity, in addition to console logs. In the next chapter, we review existing work focusing on mining console logs, as well as projects in system monitoring in general.

Chapter 7

Related Work

In the Section 7.1, we first review existing work around system monitoring and problem detection with methods other than console logs. We focus on identifying the spectrum of such work based on how much and at how fine a granularity. Then in Section 7.2, we review projects on analyzing textual logs. We group these works based on their way of modeling console logs: as raw text, as a single time series, or as separate program execution traces. We also discuss different methods of parsing console logs. Finally in Section 7.3, we summarize techniques and building blocks used in this dissertation.

7.1 System Monitoring and Problem Detection

Detecting and diagnosing problems in computer systems are useful are many contexts, especially in data centers. Obviously, problem detection requires collecting a certain amount of monitoring data. Collecting and analyzing monitoring data are both important research topics. Existing work differs fundamentally in the granularity of monitoring data required. In this section, we review previous work in system monitoring without using console logs, and we delay reviewing of console log analysis work to Section 7.2.

We present these projects as a spectrum. At one end of the spectrum, there are tools collecting coarse granularity data, such as performance counters and raw logs, and using such simple data to detect or describe problems. These kinds of tools incur the least amount of performance or maintenance overhead, but they provide the least insight into the problem itself. At the other end of spectrum, one can use source code or binary instrumentation to collect very fine grained data. These methods, though they sometimes carry high performance overhead and require significant developer efforts to implement, can help discover and diagnose subtle problems. Of course, many tools try to find a good tradeoff between the amount of data to collect and the power in diagnosing problems.

7.1.1 System monitoring tools

Passive, out-of-the-box data collection

Many data are collected by built-in mechanisms during system operation without special configurations. These data typically include performance counters, such as CPU/memory utilization, request rate, I/O statistics, as well as various types of logs (e.g. request logs, binary logs, and console logs).

Although many systems dump such traces into a text file on local disks, many other tools support distributed data collection. Simple Network Management Protocol (SNMP) [13] is the most widely deployed protocol to collect structured traces from a variety of software and devices. It supports a hierarchical name space and can be used to collect both numerical traces and less structured event traces. There are many commercial or open source tools for analyzing and visualizing SNMP data, most notably, HP OpenView (and its successors) [46], IBM Tivoli [47] and Microsoft System Center Operation Manager [71]. The predefined hierarchical namespace makes SNMP traces easy to handle, but they do not always provide the flexibility for programmers to track all data they want. This lack of flexibility is an important reason why console logs are still widely used.

OS and framework-level instrumentations

Though generic metrics such as CPU and I/O statistics can sometimes reveal interesting problems, some situations demand more detailed monitoring information. The instrumentation can be at different layers of the system, for example, at virtual machine level, OS level, programming framework level, or application level. Of course, application level instrumentation provides end-to-end detection power, but also involves significant developer or runtime overhead. In this section, we review OS and framework level instrumentation, leaving application instrumentation to the next section.

Existing work aims at replaying program execution for debugging. The tracing can be done either at virtual machine level [55] or at framework level [34]. Replay-based systems are especially useful for dealing with synchronization bugs. However, the amount of data required for replay is large, making it infeasible to keep a long history. Although recent work [2] shows that replay can be done with partial data, the performance overhead is still significant (over 5%). Another drawback is that it is often hard to know which time point during the reply really reveals "interesting" system behavior, which is essentially the same problem as finding abnormal patterns in console logs.

Modern operating systems provide mechanisms to collect both OS and process level metrics. Windows provides a large number of performance counters through the registry interface [70], while Solaris provides DTrace [12, 91], a configurable and extensible instrumentation framework. Like console logs, these data are multi-dimensional, but unlike console logs, these counters capture lower OS-level information, which sometimes do not provide enough information to detect application layer problems.

Some data requires specialized monitoring infrastructure. For example, Fonseca et al. developed XTrace [31], a framework for collecting execution path data in a distributed system. XTrace can be built into the runtime environment, such as application servers or libraries, using source code instrumentation. In contrast to the execution traces we created from console logs, the ordering of events in XTrace is guaranteed to be correct even across multiple nodes.

Stack traces and memory dumps are also widely used in debugging software. These data can be collected periodically or during a system crash. For example, Windows crash dumps are very effective in identifying common bugs in Windows [36].

Application level instrumentation

The most straightforward method of collecting application layer metrics is instrumenting the source code. Many language runtime environments provide dynamic assertions or predicate support. Predicates can be added automatically or manually. Manually added predicates are similar to console logs in the sense that they capture developers' expert knowledge. An obvious drawback of adding these predicates is increased developer overhead. Though predicates can also be generated automatically [103], generated predicates tend to be less targeted and result in a huge amount of useless data, which implies sophisticated sampling or other preprocessing to reduce the amount of data.

It is generally less desirable to tightly couple monitoring instrumentations with the source code. There are tools providing better flexibility and maintainability of instrumentation. Aspect Oriented Programming tools, such as AspectJ [5], allow operators to implement instrumentation at configurable *cut points*, such as function entrances and exits, during program execution.

Sometimes it is useful to instrument applications running on machines not easily accessible, such as thin-client applications running in a browser. Kiciman et al. proposed AjaxScope [54], a proxy-based solution to provide runtime-configurable instrumentation, including dynamic predicates, for JavaScript-based applications.

In summary, all existing tools described above provide developers and operators with a large selection of methods to collect monitoring data from different layers with different granularity. Like console logs, as more traces are collected, they quickly become infeasible to process manually. These problems all call for automated problem detection tools.

7.1.2 System problem detection with structured data

System monitoring frameworks provide information with different resolution and volume. Accordingly, there is a spectrum of existing work that makes use of the monitoring data to detect, categorize, or even debug problems. In this section, we review these projects using structured traces, in the order of increasing amount/granularity of data.

Simple numerical metrics become powerful when their correlations are considered. On the anomaly detection problem, Lakhina et al. analyzed package count data on individual links, and used a PCA based method to detect network-wide traffic anomalies [58]. Interestingly, the anomaly is only obvious when analyzing the traffic correlations on multiple links. Correlations among numerical metrics are also used to categorize similar failures.

Cohen et al. and Bodik et al. use a large collection of metrics, from OS level to application level, such as CPU, I/O and request rate and response time. They correlate these metrics with alternatively identified problematic periods (e.g. high response time or user-visible errors) in order to discover a subset of metrics whose changes best describe the root cause of the problem. As a by-product, the changes of such metrics can also be used as a description of each type of problems [19, 8].

Timestamps are also widely used in problem detection, especially in system event data or SNMP traces. Uncommon periodicity of certain event types is often a good indication of problems. Hellerstein et al. developed a novel method to mine important patterns such as message burst, message periodicity and dependencies among multiple messages from SNMP data in an enterprise network [45, 67]. There are two limitations of these kinds of methods: 1) they require accurate detection of event type, which is not always possible; 2) sometimes the periodicity is less obvious in noisy workloads, and often require ad hoc filtering.

Execution-path-based methods are very powerful. The execution paths can be obtained either through custom instrumentation or frameworks such as XTrace [31]. Path-based analysis is able to review anomalies down to individual operation level. For example, Chen et al. used clustering [17] and probabilistic context free grammars [16] to analyze execution paths in several custom instrumented systems. Our idea of message count vector feature comes from the path-based methods. In contrast, we construct execution path information solely from console logs. We do not always have correct ordering or as detailed information as traditional trace collection. We showed that with our feature design we can tolerate such inaccuracies.

Lower level performance counters can sometimes reveal high level application problems. Yuan et al. use patterns in traces of system calls to capture application layer behaviors, and are able to classify application failures, even if these failures exhibit the same symptom (e.g. slowdown) from the application perspective [101].

Many projects use periodic snapshot of system states, most notably configurations, in addition to performance traces, to diagnose problems. For example, PeerPressure [97] uses Windows Registry data to detect problems, while Dunagan et al. use changes in configuration as supplements to OS counters to detect potential security problems [22]. Kandula et al. use system configuration snapshots to capture dependencies of various components in a network system to determine the root cause of runtime anomalies [52].

When the luxury of application specific instrumentation is possible, more powerful techniques such as distributed *predicate evaluation* can be used. D^{3}S [63] demonstrated a combination of static analysis and runtime evaluation of distributed predicates to detect hard-to-find bugs in many distributed systems.

Statistical debugging [103] is a tradeoff between the large number of predicates and overhead of collecting debugging traces. It tests and collects randomly generated predicates as well as execution sequence from instrumented programs. Through many runs of the same program, abnormal executions paths can be found together with the cause of the anomaly (e.g. the abnormal predicates).

7.2 Console Log Analysis

While all console logs are collected with a very simple mechanism, which is passively collecting text streams, the analysis on console logs spans a spectrum, depending on the level of detail of information contained in logs. In this section, we first review tools for generating and collecting console logs, and then we discuss previous work in parsing console logs. In Section 7.2.3, we discuss different methods for analyzing console logs, based on different models used.

7.2.1 Console generation and collection

Traditionally, console logs are generated with standard printing statements, such as *printf* in C or *cout* stream in C++.

Because of the wide deployment of console logs, many projects target improved console log generation. Advanced logging frameworks such as log4j [39] or SLF4j [86] are developed to support flexible (or even runtime configurable) formatting, improved I/O performance and allow multiple repositories. The development of these tools decoupled log generation from collection, and greatly improved manageability of logging in large scale systems. In this project, we assume programmers use such frameworks for log generation instead of using printing statements directly, so we can get additional information such as timestamps and thread IDs associated with each message. We do not modify the framework or the logger configuration.

Console logs are typically collected using *syslog* [64]. Notice that syslog, besides requiring a timestamp and the module name where log is generated, does not require any specific format for the free text part of the log message, and thus to operators it is the same as console logs. Syslogs are traditionally collected using *syslogd*, either on the local node or over the network. More sophisticated implementations focus on improving availability and security [7]. On the other hand, as syslog implementation requires each log message to be sent to the syslog server through remote procedure call (RPC) as they are generated, it is hard to scale to a very large scale cluster.

Chukwa [10] implemented a decentralized log collecting system that stores log archives on Hadoop File System. This design not only provides reliability to log archives, but also makes accessing historical logs efficient with map-reduce programming tools [9]. Our problem detection framework can directly handle logs stored in Chukwa.

The large variety of tools discussed reflects the fact that console logs are still widely used in practice. On the other hand, the lack of analysis tools makes the vast amount of textual logs less useful, which is the major problem we aim to solve in this project.

7.2.2 Console log parsing

Almost all existing automated console log analysis work uses concepts similar to message types and variables discussed in our project. A questionable assumption in previous work is that message types can be detected accurately. Both [45, 67] use manual type labels from SNMP data, which are not generally available in console logs.

Most projects use simple heuristics—such as removing all numeric values and IP-address-like strings—to detect message types [99, 61]. These heuristics are not general enough. If the heuristics fail to capture some relevant variables, the resulting number of message types can be in the tens of thousands [61].

Several efforts use frequent item set mining, a technique from data mining, to find frequently appearing string segments in order to detect message type [95, 89, 96]. In order to improve accuracy, many heuristics are added to normal frequent item set algorithms, such as considering the position of a certain word. The authors of [30] use more advanced clustering and association rules as well as scoring methods from information retrieval to extract message templates for log parsing. IPLoM [68] uses a series of heuristics to iteratively capture the differences of similar log messages to determine message types. Fu et al. use a similar iterative method to find message types [32].

This class of methods works well on messages types that occur many times in log, but it cannot handle rare message types that are likely to be related to the runtime problems we are looking for in this research. In our approach, we combine log parsing with source code analysis to get accurate message type extraction, even for rarely seen message types.

7.2.3 Using console logs for problem detection

Console logs as free text

Traditionally, operators use *grep* or write ad-hoc scripts (mostly in Perl or Python) for console log monitoring and analysis.

One popular improvement to these ad-hoc scripts is rule-based systems. The most popular ones include Logsurfer [80], Swatch [42], and OSSEC [76]. In these systems, operators need to specify two types of rules: regular expressions that extract certain textual patterns from log messages, and rules that perform simple aggregations on the patterns extracted.

The problem with rule-based systems is that the rules, like scripts, are usually hard devise, and even harder to maintain. Previous research [75] shows that most of the naive rules either do not reveal the real problem or have too many false alarms.

Another direction of improving ad hoc scripts is adopting full text search tools, such as Splunk [87], a commercially available tool. Although these tools bring significant improvements in terms of speed and ease-of-use over scripting, they still require operators to provide key words for searching, even though unfortunately the key word selection process is often beyond operators' knowledge.

There is no fundamental difference between either rule-based systems or Splunk and traditional ad hoc scripting in that all of them models logs as collections of English words and only consider the *textual* properties of logs. As [75] suggests, however,the semantics of words are not reliable indicators of system problems, which is also shown in the "read exception" example discussed in Section 4.3. Recall that in that example the "exception" was actually normal behavior. In contrast, our approach extracts information about program objects from log messages, and our detection is based on event traces related to those objects, rather than on textual properties.

Console logs as event streams

Using only message type information, existing work treats the entire log as a single sequence of events and applies time series analysis methods.

One focus of analyzing temporal properties is to reduce repeated alarms in monitoring systems. Liang et al. greatly reduced number of repeated alarms on hardware failures from a supercomputer by correlating similar messages from multiple components that occur around the same time [60]. Lim et al. analyzed a large scale enterprise telephony system log with multiple heuristic filters, such as number of messages in a time window and change rate of the message count, to reduce the number of alerts before actual failures [61]. These methods reduce the number of repeating events that humans must read, but do not reduce the number of *types of* events humans need to handle manually, which is the focus of our project.

Another focus for analyzing temporal properties of console logs is finding temporal correlations of some message types. Yamanishi et al. model syslog sequences as a mixture of Hidden Markov Models (HMM) in order to find messages that are likely to be related to critical failures [99]. However, treating a log as a single time series does not perform well in large scale clusters with multiple independent processes that generate interleaved logs. The model becomes overly complex and parameters are hard to tune with interleaved logs. Our analysis leverages additional information such as identifiers to recover the original sequences, resulting much cleaner feature vectors.

Console logs as program state transitions

Recent work, including ours, models console logs as a number of interleaving execution traces. Fu et al. compute a finite state machine (FSM) from console log stream, each message type representing a (somewhat abstract) program state [32]. The expected sequence of messages (state transitions) is captured by the FSM, and the model is augmented with expected transition times. Comparing to our method, though FSM is able to capture more information than the message count vector, it is highly sensitive to noise (such as the random reordering, inaccurate timestamps and high variations in durations, as described in Section 5.1), especially in a large scale system with highly variable workloads.

7.3 Techniques Used in This Project

We get many of our ideas and techniques from existing research in systems, machine learning, and information retrieval, although to the best of our knowledge our technique represents the first time these methods are applied to console log analysis.

Anomaly detection

PCA-based anomaly detection method was originally developed in multivariate process control [23], and has been applied to many areas. The most relevant one is network anomaly detection. Lakhina et al. analyzed package count data on individual links, and used PCA based method to detect network-wide traffic anomalies [58]. This method works both on networking data and in our console log data because both types of data share intrinsic low dimensionality, which we discussed in Section 4.2. To the best of our knowledge, our project is the first time PCA method was used on system logs and execution paths in general.

Kernel methods have been used in many areas in machine learning such as classification, clustering, regression as well as anomaly detection to handle non-linear patterns in data. Kernels capture the *similarity* of multiple kinds of data, including discrete and structured data [33, 49]. We do not explicitly use a kernel method. Instead, we pre-process message count vectors with *term frequency / inverse document frequency* (TF/IDF), a well-established method in information retrieval [83, 78], and we use *cosine similarity* to normalize the vectors. We show that it is similar to the multinomial product kernel discussed in [49]. This normalization improves results significantly because it is a better and more accurate way of capturing differences among message count vectors.

Frequent pattern mining

Within the vigorous research area of frequent pattern mining [40], which has been an active research field in data mining for over a decade, a variety of efficient algorithms has been proposed in different research frontiers [1, 41, 88, 102, 79]. We are particularly interested in sequential pattern mining techniques, which mine frequently occurring ordered subsequences as patterns. For example, Generalized Sequential Patterns (GSP) [88] is a representative Apriori-based algorithm; SPADE [102] is a vertical format-based mining method; PrefixSpan [79] is a pattern-growth approach to sequential pattern mining. We extend the techniques to address the unique challenges of our problem described in Section 5.2. The major difficulties for us, which are not addressed in these general algorithms, are: 1) "transactions" (sessions) interleave in our data. Thus we need to simultaneously segment a sequence into sessions and mine patterns. 2) We only observe partial orders of the events in the sequences.

Frequent pattern mining techniques have also been used to analyze words in messages to understand the structure of console logs [95, 96] and to discover recurring runtime execution patterns in the Linux kernel [59]. In contrast, we use frequent pattern mining to message types that frequently appear together, rather than patterns of individual words.

Using textual information in software development

Software development involves textual information other than console logs. Previous work demonstrated that by leveraging source code information, the best maintained and structured, one can make the unstructured text much easier to handle. The Javadoc system explicitly allows developers to link the API documentation to the corresponding source code segments [57]. This connection enables tools leveraging source code information to help developers better browse or search [25], validate and evaluate [74], and share [21] API documentation. Combining natural language processing techniques with static source code analysis, Tan et al. proposed a novel approach to detect inconsistencies between textual comments and program logic [93].

Our idea is similar in that we can make textual information machine understandable by leveraging source code. The "link" between console log message and the source code is *implicitly* defined (by the string constants), rather than explicitly specified as in the Javadoc case. For this reason, a significant part of our approach is to automatically recover this link. Also, console log messages are far less meaningful to human reader than documentation, which is why we use them as execution sequences rather than natural language sentences.

System building

From the system design perspective, our system is built on many open source tools. We used Eclipse Java Development Tooling (JDT) [82] and C++ Development Tooling (CDT) [38] to implement our source code analyzer and Lucene [44] to match message templates to actual messages. We used Hadoop map-reduce [9] to scale our experiment to many nodes. We used RapidMiner [72], a machine learning and data mining framework to generate the decision trees. In the online detection implementation, we also employed basic streaming database concepts, such as stream filtering, aggregation and group-by [6, 15] and implemented each components in our system as a stream processor.

7.4 Summary

Monitoring and diagnosing problems in distributed systems is such an important and difficult topic that much work has been done in collecting monitoring data, as well as automatically discovering problems. Traditionally, the methods used on console log analysis are quite different from those methods used on other system diagnosis work, due to the free text nature of console logs. Our work, in contrast, uses source code analysis to parse console logs as the first step, which transforms the free text logs into semi-structured representation. This representation potentially makes many of these techniques applicable on these rich sources of information.

In the next chapter, we discuss potential research directions in analyzing console logs and applying similar techniques to different kinds of data in system development and operation in general.

Chapter 8

Future Directions

In this dissertation, we show that the widely-used but commonly-ignored console logs can be automatically turned into a powerful source of information for system problem diagnosis. Specifically, the parsing and feature creation techniques allow many machine learning algorithms to be applied to console logs.

As future work, we first discuss challenges in allowing anomaly detection algorithms to use extra information and become more robust. Then we show the necessary changes to current logging frameworks to make console logging more useful for debugging. Finally, we discuss potential applications of our methodology to other aspects of program development.

8.1 Anomaly Detection and Beyond

The anomaly detection techniques discussed in previous chapters can be further improved by analyzing additional information sources, such as structured traces, logs from related applications, and operator feedback.

One interesting direction is to analyze logs from multiple layers or subsystems of the entire application stack. In a large scale Internet service consisting of multiple independently developed (e.g. open source) subsystems, the logs from each subsystem can be completely different. However, they are implicitly correlated because of the application logic. Discovering the implicit correlations is the key to discovering many hard-to-debug, cross-layer problems.

It is also interesting to bring human operators back into the debugging loop by allowing operators to give feedback to the detector and incorporating this feedback to refine the detection. It is especially challenging to design a human-friendly interactive learning process.

Combining console log information with other structured traces is also an interesting future direction. It is especially useful to find the messages that could explain some "mysteries" in changes of structured traces. Our Darkstar case study is a successful application, but discovering correlations other than co-occurrence in time requires more research on anomaly detection algorithms.

Another direction is to make the anomaly detection algorithm more robust in handling incomplete logs. Sometimes it is impossible to collect and archive a perfect set of logs from a very large scale system consisting of tens of thousands of nodes. Some messages might be corrupted. Unlike the noise we discussed in Chapter 5, these corruptions cause messages to be missing or become completely unrecognizable. The corruption can be random (e.g. due to software/disk errors) or deterministic (e.g. due to physical failure of nodes or network connection) [75]. Tolerating these corruptions while and still providing an accurate problem detection result is a challenging machine learning problem.

Beyond problem detection, we can use console log data to support other important tasks in data center management such as resource planning and job scheduling. Of course, these tasks would require different sets of features and machine learning algorithms, but the general methodology is similar and the methods of transforming the free-text console log to the structured features are still applicable.

With some improvement to the logging framework, we can do even more with machine learning. In the next section, we first discuss our proposed improvement to the logging framework and then discuss more potential machine learning applications.

8.2 Tools to Improve Console Log

Up until this point, our method has assumed using existing console logs “as is”. Though this assumption is convenient for developers and operators, the power of problem detection is limited by the information contained in the console logs. As a vital future direction, we believe it is important to provide automatic or semi-automatic tools to help developers create better console logs that can be more useful in problem diagnosis.

We describe potential improvements to both the mechanism and policy for generation and collection of console logs.

Mechanism: Improving console log generation

The console log provides a highly flexible way to report monitoring data in a wide range of granularity, but the flexibility comes at a cost. Compared to well-defined binary logging formats, one needs to serialize and concatenate message variables into a text string. Even worse, certain logging libraries require the construction of the log message even if the particular message is suppressed by the logging level configuration [39].

The other problem is that the runtime control of which messages to print is not flexible enough. It is only possible to configure logging by levels (e.g. WARNING, FATAL), or by program modules, while a fully automatic problem detection system would benefit from the ability to control each individual message type. Programming language and runtime-system level support can address these two problems.

We can build our log parsing techniques discussed in Chapter 3 directly into the compiler to help the compiler understand message types and message variables. With this information, the compiler can emit optimized code to avoid the expensive and unnecessary string concatenations. The compiler can even emit code so that the console logs are collected in a raw binary format instead of text. This improvement also eliminates the computation overhead of the log parsing step.

In addition, if the compiler has knowledge about message types, it is possible to support mechanisms to turn on/off any individual message type at runtime in an efficient way, traditionally not possible in logging libraries. Even better, it is easy to dynamically reconfigure the destination of each individual message such as in memory buffer, local disk, or an event processing system. With multiple destinations of different overhead, the automated problem detector can keep a short but detailed log history at a relatively low cost.

In practice, it might take a long time to deploy a new mechanism into popular programming languages such as Java. To support incremental deployment, we can implement such mechanisms with binary transformation [77], or higher abstractions such as Aspect Oriented Programming [5].

Another improvement to console logs is to capture concurrency-related bugs. Console logs are not designed with concurrency-support in mind. Though many recent logging libraries are thread-safe, the thread-safety only guarantees that two messages do not interleave with each other. With the increasing importance of implementing parallel applications, logs should record necessary metadata to help reveal concurrency-related bugs.

Policy: What should we log?

Improving log generation efficiency is not enough. The major "overhead" of logging is the cost of generating and archiving a vast amount of messages that are completely useless. Operators often run into the dilemma of accepting the overhead versus risking not being able to capture certain problems. Two most difficult questions when designing a logging system are 1) how detailed the monitoring data should be, and 2) how long a history to keep.

Online machine learning techniques combined with the improved logging mechanisms can help solve these two problems. We can use online machine learning techniques to capture and filter out the normal patterns (thus the unimportant parts of the log), and automatically turn off messages/traces that are highly unlikely to indicate any problems. On the other hand, once a potentially anomaly is detected, the detector could turn on related message types, or change the destination of certain message types, such as dumping in-memory log segments to disk. As we discussed earlier, the ability to quickly detect potential problems and reconfigure logging can be very useful to capture bugs that occur only on certain inputs: we can avoid logging normal operations while providing detailed logging for problematic situations.

8.3 Beyond Console Logs

Looking beyond console logs, our techniques of combining program analysis, information retrieval and machine learning are also useful in managing other information related to software development and operation.

The goal of software engineering is to develop and operate high-quality software. During this process, there is a lot of information generated and designed to be used by different roles in the process. For example, the developers usually use comments, documentation, version control logs, makefiles, unit tests, and continuous build tools, while operators are more familiar with operational documentation, deployment scripts, trouble tickets, console logs, and various monitoring tools.

In a modern Internet service, the distinction between developers and operators is becoming less obvious, and the amount of information can quickly become hard to manage as the system gets more complex. Existing solutions use an information retrieval system to manage these data; however, the lack of structured information usually makes such systems hard to use, and even harder to automate. Since source code is the best maintained and easiest to analyze information in the entire cycle, we believe that by leveraging source code information, we can make many powerful machine learning algorithms applicable to these unstructured data and provide much better tools for both developers and operators.

For example, an interesting future direction is to design a tool to help maintain documentation, especially operator-oriented documentation, while source code can change rapidly. We can use machine learning algorithms to capture the "patterns" of misconfigurations or bad inputs that caused problems. These patterns can provide insights to developers so they can produce better documentation to avoid confusing operators. We could also suggest under-tested cases to developers based on problems discovered in large scale deployments.

Chapter 9

Conclusion

Despite the fact that console logs are a rich information source for system monitoring and diagnosis, they are not fully utilized by system operators today. In this dissertation, we present a general approach to automatically process console logs to detect problems in large scale Internet service systems.

Our basic assumption is to work on console logs *as is*, without changing the system under study. Comparing to other system monitoring data, there are two major challenges working on console logs: the size and the unstructured free text form. We address these two challenges with a general four-step methodology: log parsing, feature creation, machine learning, and visualization. Each step reduces the amount of data to process, and more importantly, each step explores the intrinsic structures and statistical properties of console logs and makes logs more structured.

- The message parsing step automatically captures the structure of *individual messages* by analyzing the print statements in program source code. By transforming a free text message into a semi-structured form—the message type and message variables—we can eliminate all complexities of dealing with free text in further steps.

- Designing features by grouping relevant messages, we explore the structure of *multiple messages*. There are different ways to group console logs. In this dissertation, we discussed two types of features: state ratio vectors describing the global state of the system, and message count vectors capturing sequences of operations.

- The number of feature vectors can still be overwhelming for human operators to examine. We use data mining and machine learning techniques to find the *statistical patterns* from the vast amount of data and detect anomalies automatically. This step *reduces the amount of data* an operator needs to examine by more than 99%.

- Finally, we put the numerical features and detection results back into the context of logs, and create an *intuitive diagram* to allow operators not familiar with machine learning techniques to easily interpret the result. The visualization step reduces the amount of alarm information to a single page, a more human friendly form.

As a design philosophy, we focus on leveraging domain knowledge about distributed systems to design meaningful features and reduce noise, rather than relying on complex machine learning algorithms to tolerate unnecessary noise. For example, instead of trying to "guess" the structure of log messages statistically, we make use of program analysis tools to extract them accurately from program source code or even machine binaries (Chapter 3). As another example, by grouping relevant message using identifers, we can eliminate all the complexities dealing with concurrent events, allowing us to use simple and efficient algorithms such as PCA (Chapter 4). As another example, we observed that in production systems, most of the identifiers go through a similar execution path. This observation greatly simplifies the online problem detection algorithm, allowing us to use the two-stage design (Chapter 5).

Console log analysis is a very practical problem, and all data sets used in our evaluations come from real-world programs. The size of these data sets ranges from a few megabytes to hundreds of gigabytes. All of our implementations are able to scale up and down to handle logs from a single node to a cluster of thousands of nodes. There are two implementations for most of our processing steps: as a batch job or as an online stream processor.

Our experiments show that we can extract new and useful insights from all systems we studied. The detection results can be useful for system operators and developers in two ways.

1. In some cases, the anomalies are otherwise hard to discover without using console logs. For example, one type of anomaly in Hadoop, which indicates a software bug, is very hard to discover without analyzing sequences of log messages (Section 4.3.2).

2. Sometimes it is easy to notice a problem, but console log analysis is still useful to help find the cause of the problem. In the case studies on Darkstar and Google's production logs, it is easy to notice the performance degradation, but not until the logs are analyzed can we discover the likely source of the problem (Section 4.3.1 and Section 6.2).

Console logs have been used since the dawn of programming, and generations of programmers rely on the logs to monitor and debug their systems. By exploring the intrinsic structures and patterns in console logs, our work turns console logs into a powerful monitoring system for problem detection, even in very large systems.

Bibliography

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 1994 international conference on very large data bases (VLDB94)*, Santiago, Chile, 1994.

[2] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, New York, NY, 2009.

[3] Amazon Web Services. Amazon elastic compute cloud developer guide. `http://aws.amazon.com/`, 2008.

[4] A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, second edition, 2002.

[5] P. Avgustinov, E. Hajiyev, N. Ongkingco, O. de Moor, D. Sereni, J. Tibble, and M. Verbaere. Semantics of static pointcuts in AspectJ. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, 2007.

[6] S. Babu and J. Widom. Continuous queries over data streams. In *Proceedings of International Conference on Management of Data (SIGMOD 2001)*, Sept. 2001.

[7] Balabit.com. Distributed syslog architectures with syslog-ng premium edition. `http://www.balabit.com`.

[8] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen. Fingerprinting the datacenter: Automated classification of performance crises. In *Proceedings of EuroSys'10*, Paris, France, 2010.

[9] D. Borthakur. The hadoop distributed file system: Architecture and design. `http://hadoop.apache.org`, 2007.

[10] J. Boulon, A. Konwinski, R. Qi, A. Rabkin, E. Yang, and M. Yang. Chukwa: a large-scale monitoring system. Chicago, IL, Oct. 2008.

[11] E. Bruneton. Asm 3.0: A Java bytecode engineering library. `http://download.fr.forge.objectweb.org/asm/asm-guide.pdf`.

[12] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of USENIX Annual Technical Conference(ATEC'04)*, Boston, MA, 2004.

[13] J. Case, M. Fedor, M. Schoffstall, and J. Davin. Rfc 5343: Simple network management protocol. `http://tools.ietf.org/html/rfc5343`, Aug. 1988.

[14] I. Chakravarti, R. Laha, and J. Roy. *Handbook of Methods of Applied Statistic*, volume I. John Wiley and Sons, 1967.

[15] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data (SIGMOD'03)*, San Diego, CA, 2003.

[16] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, San Francisco, CA, 2004.

[17] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN'02)*, Washington, DC, 2002.

[18] A. Clauset, C. Shalizi, and M. Newman. Power-law distributions in empirical data. *SIAM Review*, 2009.

[19] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *Proceedings of the twentieth ACM symposium on Operating systems principles (SOSP'05)*, Brighton, UK, 2005.

[20] M. H. DeGroot and M. J. Schervish. *Probability and Statistics*. Addison-Wesley, 3rd edition, 2002.

[21] U. Dekel. eMoose: a memory aid for software developers. In *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications (OOPSLA Companion '08)*, Nashville, TN, 2008.

[22] J. Dunagan, A. X. Zheng, and D. R. Simon. Heat-ray: combating identity snowball attacks using machinelearning, combinatorial optimization and attack graphs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP '09)*, Big Sky, MT, 2009.

[23] R. Dunia and S. J. Qin. Multi-dimensional fault diagnosis using a subspace approach. In *Proc. of American Control Conference (ACC'97)*, Albuquerque, NM, 1997.

[24] C. Eagle. *The IDA Pro Book: the unofficial guide to the world's most popular disassembler*. No Starch Press, San Francisco, CA, 2008.

[25] Eclipse.org. Eclipse integrated development environment. http://www.eclipse.org, Jan 2010.

[26] Edgewall.org. Trac: integrated SCM and project management. http://trac.edgewall.org/, Jan 2010.

[27] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *Proceedings of SIGCOMM'99*, Cambridge, MA, 1999.

[28] E. Farchi, G. Kliot, Y. Krasny, and A. Krits. Effective testing and debugging techniques for a group communication system. In *Proceedings of Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, Washington, DC, 2005.

[29] R. Feldman and J. Sanger. *The Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data*. Cambridge Univ. Press, 12 2006.

[30] K. Fisher, D. Walker, K. Q. Zhu, and P. White. From dirt to shovels: fully automatic tool generation from ad hoc data. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, San Francisco, CA, 2008.

[31] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. Xtrace: A pervasive network tracing framework. In *Proceedings of Symposium on Networked Systems Design and Implementation (NSDI'07)*, Cambridge, MA, 2007.

[32] Q. Fu, J.-G. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining (ICDM'09)*, Washington, DC, 2009.

[33] T. Gärtner. A survey of kernels for structured data. *SIGKDD Explor. Newsl.*, 5(1), 2003.

[34] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference (ATEC'06)*, Boston, MA, 2006.

[35] G. Giuseppini. *Microsoft log parser toolkit*. Syngress Publishing, 2005.

[36] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: ten years of implementation and experience. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, Big Sky, MT, 2009.

[37] Google Inc. Google App Engine SDK. http://code.google.com/appengine/, Jan 2010.

[38] E. Graf, G. Zgraggen, and P. Sommerlad. Refactoring support for the C++ development tooling. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, Montreal, Canada, 2007.

[39] C. Gulcu. *Short introduction to log4j*, March 2002. `http://logging.apache.org/log4j`.

[40] J. Han, H. Cheng, D. Xin, and XifengYan. Frequent pattern mining: current status and future directions. *Data Mining and Knowledge Discovery*, 15(1), 2007.

[41] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD international conference on management of data (SIGMOD'00)*, Dallas, TX, 2000.

[42] S. E. Hansen and E. T. Atkins. Automated system monitoring and notification with Swatch. In *LISA '93: Proceedings of the 7th USENIX conference on System administration*, Monterey, CA, 1993.

[43] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition*. Springer Series in Statistics. Springer, 2nd edition, September 2010.

[44] E. Hatcher and O. Gospodnetic. *Lucene in Action*. Manning Publications Co., Greenwich, CT, 2004.

[45] J. Hellerstein, S. Ma, and C. Perng. Discovering actionable patterns in event data. *IBM Sys. Jour*, 41(3), 2002.

[46] HP. HP OpenView performance manager. `http://www.managementsoftware.hp.com/`, Jan 2010.

[47] IBM. Tivoli software. `http://www.ibm.com/software/tivoli/`, Jan 2010.

[48] J. E. Jackson and G. S. Mudholkar. Control procedures for residuals associated with principal component analysis. *Technometrics*, 21(3), 1979.

[49] T. Jebara, R. Kondor, and A. Howard. Probability product kernels. *Journal of Machine Learning Res.*, 5, 2004.

[50] W. Jiang, C. Hu, S. Pasupathy, A. Kanevsky, Z. Li, and Y. Zhou. Understanding customer problem troubleshooting from storage system logs. In *FAST '09: Proccedings of the 7th conference on File and storage technologies*, San Francisco, CA, 2009.

[51] I. Jolliffe. *Principal Component Analysis*. Springer, 2002.

[52] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl. Detailed diagnosis in enterprise networks. In *SIGCOMM '09: Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, Barcelona, Spain, 2009.

[53] B. W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 1988.

[54] E. Kiciman and B. Livshits. AjaxScope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, Stevenson, WA, 2007.

[55] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *ATEC '05: Proceedings of USENIX Annual Technical Conference*, Berkeley, CA, 2005.

[56] C. Kit, H. Pan, and H. Chen. Learning case-based knowledge for disambiguating chinese word segmentation: a preliminary study. In *Proceedings of the first SIGHAN workshop on Chinese language processing*, Morristown, NJ, 2002. Association for Computational Linguistics.

[57] D. Kramer. API documentation from source code comments: a case study of Javadoc. In *SIGDOC '99: Proceedings of the 17th annual international conference on Computer documentation*, New Orleans, LA, 1999.

[58] A. Lakhina, M. Crovella, and C. Diot. Diagnosing network-wide traffic anomalies. In *Proc. ACM SIGCOMM*, Portland, OR, 2004.

[59] C. LaRosa, L. Xiong, and K. Mandelberg. Frequent pattern mining for kernel trace data. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, Fortaleza, Brazil, 2008.

[60] Y. Liang, A. Sivasubramaniam, and J. Moreira. Filtering failure logs for a BlueGene/L prototype. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks*, Washington, DC, 2005.

[61] C. Lim, N. Singh, and S. Yajnik. A log mining approach to failure analysis of enterprise telephony systems. In *Proceedings of the 2008 International Conference on Dependable Systems and Networks (DSN'08)*, June 2008.

[62] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[63] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D^{3}S: debugging deployed distributed systems. In *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, San Francisco, CA, 2008.

[64] C. Lonvick. RFC3164 - the BSD syslog protocol. `http://www.faqs.org/rfcs/rfc3164.html`, August 2001.

[65] P. Louridas, D. Spinellis, and V. Vlachos. Power laws in software. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*, volume 18, 2008.

[66] M. Lutz. *Programming Python.* O'Reilly Media, Inc., 2006.

[67] S. Ma and J. L. Hellerstein. Mining partially periodic event patterns with unknown periods. In *Proceedings of the 17th International Conference on Data Engineering (ICDE'01)*, Washington, DC, 2001.

[68] A. A. Makanju, A. N. Zincir-Heywood, and E. E. Milios. Clustering event logs using iterative partitioning. In *KDD '09: Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, Paris, France, 2009.

[69] C. D. Manning, P. Raghavan, and H. Schtze. *Introduction to Information Retrieval.* Cambridge University Press, 2008.

[70] Microsoft. Consuming counter data. `http://msdn.microsoft.com/en-us/library/aa371903(VS.85).aspx`, Jan 2010.

[71] Microsoft. System center operation manager. `http://www.microsoft.com/systemcenter/operationsmanager`, Jan 2010.

[72] I. Mierswa, M. Wurst, R. Klinkenberg, M. Scholz, and T. Euler. YALE: rapid prototyping for complex data mining tasks. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, Philadelphia, PA, 2006.

[73] MoinMoin Team. The MoinMoin wiki engine. `http://moinmo.in/`, Jan 2010.

[74] S. N. I. Mount, R. M. Newman, R. J. Low, and A. Mycroft. Exstatic: a generic static checker applied to documentation systems. In *SIGDOC '04: Proceedings of the 22nd annual international conference on Design of communication*, Memphis, TN, 2004.

[75] A. Oliner and J. Stearley. What supercomputers say: A study of five system logs. In *Proceedings of the 2007 International Conference on Dependable Systems and Networks (DSN'07)*, Edinburgh, UK, 2007.

[76] OSSEC.org. *OSSEC Manual*, 2008.

[77] OW2 Consortium. ASM website. `http://asm.ow2.org/`, Apr 2010.

[78] K. Papineni. Why inverse document frequency? In *NAACL '01: Second meeting of the North American Chapter of the Association for Computational Linguistics on Language technologies*, Pittsburgh, PA, 2001.

[79] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. PrefixSpan: mining sequential patterns efficiently by prefix-projected pattern growth. In *ICDE '01: Proceedings of the 17th International Conference on Data Engineering*, Heidelberg, Germany, 2001.

[80] J. E. Prewett. Analyzing cluster log files using logsurfer. In *Proceedings of Annual Conference on Linux Clusters*, 2003.

[81] K. Rieck, P. Laskov, and S. Sonnenburg. Computation of similarity measures for sequential data using generalized suffix trees. In *Advances in Neural Information Processing Systems 19*. MIT Press, Cambridge, MA, 2007.

[82] T. Sager, A. Bernstein, M. Pinzger, and C. Kiefer. Detecting similar java classes using tree algorithms. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, Shanghai, China, 2006.

[83] G. Salton and C. Buckley. Term weighting approaches in automatic text retrieval. Technical report, Cornell, Ithaca, NY, 1987.

[84] B. Schölkopf, A. J. Smola, and K.-R. Müller. Kernel principal component analysis. In *Advances in kernel methods: support vector learning*. MIT Press, 1999.

[85] E. Siever, S. Figgins, and A. Weber. *Linux in a Nutshell*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2003.

[86] SLF4j Team. SLF4J user manual. `http://www.slf4j.org/`, 2008.

[87] Splunk Inc. Splunk user guide. `http://www.splunk.com/`, Sept 2008.

[88] R. Srikant and R. Agrawa. Mining sequential patterns: generalizations and performance improvements. In *EDBT '96: Proceedings of the 5th International Conference on Extending Database Technology*, Avignon, France, 1996.

[89] J. Stearley. Towards informatic analysis of syslogs. In *LUSTER '04: Proceedings of the 2004 IEEE International Conference on Cluster Computing*, Washington, DC, 2004.

[90] Sun Microsystems. Project darkstar. `http://www.projectdarkstar.com`, 2008.

[91] Sun Microsystems. Solaris dynamic tracing guide. `http://docs.sun.com/app/docs/doc/817-6223`, 2008.

[92] J. Tan and et al. SALSA: Analyzing logs as StAte machines. In *Proceedings of USENIX Workshop on Analysis of System Logs (WASL'08)*, 2008.

[93] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /*icomment: bugs or bad comments?*/. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, Stevenson, WA, 2007.

[94] TIOBE Software. Tiobe programming community index. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, April 2010.

[95] R. Vaarandi. A data clustering algorithm for mining patterns from event logs. *Proceedings of the 2003 IEEE Workshop on IP Operations and Management (IPOM'03)*, 2003.

[96] R. Vaarandi. A breadth-first algorithm for mining frequent patterns from event logs. In *INTELLCOMM*, volume 3283. Springer, 2004.

[97] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with peerpressure. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, San Francisco, CA, 2004.

[98] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations.* Morgan Kaufmann, 2000.

[99] K. Yamanishi and Y. Maruyama. Dynamic syslog mining for network failure monitoring. In *KDD '05: Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, Chicago, IL, 2005.

[100] Y. Yao and K. T. Lua. Splitting-merging model of chinese word tokenization and segmentation. *Nat. Lang. Eng.*, 4(4), 1998.

[101] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma. Automated known problem diagnosis with event traces. *SIGOPS Operating System Review*, 2006.

[102] M. J. Zaki. Spade: an efficient algorithm for mining frequent sequences. *Machine Learning*, 42, 2004.

[103] A. Zheng and et al. Statistical debugging: Simultaneous isolation of multiple bugs. In *ICML '06: Proceedings of the 23rd international conference on Machine learning*, 2006.