Invited Applications Paper

Detecting Large-Scale System Problems by Mining Console Logs

Wei Xu XUW@CS.BERKELEY.EDU
EECS Department, UC Berkeley

Ling Huang LING.HUANG@INTEL.COM
Intel Labs Berkeley

Armando Fox FOX@CS.BERKELEY.EDU
EECS Department, UC Berkeley

David Patterson PATTRSN@CS.BERKELEY.EDU
EECS Department, UC Berkeley

Michael I. Jordan JORDAN@CS.BERKELEY.EDU
EECS and Statistics Department, UC Berkeley

Abstract

Surprisingly, console logs rarely help operators detect problems in large-scale datacenter services, for they often consist of the voluminous intermixing of messages from many software components written by independent developers. We propose a general methodology to mine this rich source of information to automatically detect system runtime problems. We use a combination of program analysis and information retrieval techniques to transform free-text console logs into numerical features, which captures *sequences* of events in the system. We then analyze these features using machine learning to detect operational problems. We also show how to distill the results of our analysis to an operator-friendly one-page decision tree showing the critical messages associated with the detected problems. In addition, we extend our methods to online problem detection where the sequences of events are continuously generated as data streams.

1. Introduction

Today's large-scale Internet services run in large server clusters in data centers and cloud computing environments. These system architectures enable highly scalable Internet services at a relatively low cost. However, detecting and diagnosing problems in such systems bring 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 (Lakhina et al., 2004; Cohen et al., 2005; Bodik et al., 2010) to system events (Hellerstein et al., 2002; Ma & Hellerstein, 2001) to more detailed tracing such as execution paths (Chen et al., 2002; Chen & Brewer, 2004). However, console logs, the debugging information built into almost every piece of software, are rarely studied by either operators or the research community.

Since the dawn of programming, developers have used everything from *printf* to complex logging and monitoring libraries (Fonseca et al., 2007; Gulcu, 2002) 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,

Appearing in *Proceedings of the 27th International Conference on Machine Learning*, Haifa, Israel, 2010.

part operator, and charged with fixing the problem—are usually not the people who chose what to log or why. 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 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 (Jiang et al., 2009; Oliner & Stearley, 2007). Rule-based processing (Prewett, 2003) 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. Instead of asking users to search, we provide tools to automatically find "interesting" log messages. Our goal is to find the "needles in the haystack" that might indicate operational problems, without any manual input.

Related work. There are two widely used models of console logs: as a collection of English terms (Stearley, 2004; Vaarandi, 2004; Splunk, 2008) or as a single sequence of repeating events (Hellerstein et al., 2002; Ma & Hellerstein, 2001; Yamanishi & Maruyama, 2005; Lim et al., 2008). These methods, however, do not perform well in large-scale systems with multiple independent processes that generate interleaved logs. We instead model console logs as a number of interleaving execution traces by pre-grouping relevant messages. This grouping process makes it possible to obtain useful results with simple, efficient machine learning algorithms. More similar to our method are the path-based problem detection approaches such as Pinpoint (Chen & Brewer, 2004; Fonseca et al., 2007), but these methods have required custom structured traces.

We also improve existing log parsing methods, which rely on repeating textual patterns in historical logs (Vaarandi, 2003; Fisher et al., 2008; Makanju et al., 2009). 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. In our approach, we combine log parsing with source code analysis to get accurate message type extraction, even for rarely seen message types.

Beyond console logs, machine learning techniques have been widely used to help computer system or network operations. Much work has been done for profiling end-hosts and networks (Xu et al., 2005; Karagiannis et al., 2007), detecting anomalous events and other intrusions (Lakhina et al., 2004; Ye et al., 2007), finding root causes for operational problems (Cohen et al., 2005; Bodik et al., 2010), predicting performance and resource utilization (Ganapathi et al., 2009), as well as pinpointing application bugs (Li et al., 2004; Zheng et al., 2006) and configuration problems (Wang et al., 2004). Our work is different from these techniques mainly because they analyze aggregate data while we detect anomalies from sequences of individual operations.

Our Contributions. We propose a general framework that is based on using a combination of program analysis, information retrieval and machine learning techniques to build a fully automatic problem detection system using console log information. Specifically, our contributions include: (1) a general methodology for automated console log processing, (2) online problem detection with message sequences, and (3) system implementation and evaluation on real world systems.

This paper summarizes the highlights of our log mining techniques, and readers may refer to Xu et al. (2009b) and Xu et al. (2009a) for more details.

2. 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 machine learning algorithms. The following four key insights lead to our solution to this problem.

Insight 1: 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 source code. Consider the simple console log excerpt and the source code that generated it in Figure 1. Intuitively, it is easier to recover the log's hidden "schema" using the source code information (especially for a machine). Our method leverages source code analysis to recover the inherent 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 ad-

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

```
1  CLog.info("starting: " + txn);
2  Class Transaction {
3    public String toString() {
4      return "xact " + this.tid +
5        " is " + this.state;
6    }
7  }
```

Figure 1. Top: two lines from a simple console log. Bottom: Java code that could produce these lines.

Table 1. State variables and identifiers

Variable	Examples	Distinct values
Identifiers	`transaction_id` in Darkstar; `block_id` in Hadoop FS; `cache_key` in Apache server; `task_id` in map-reduce.	many
State Vars	Transaction stages in Darkstar; Server names in Hadoop; HTTP status code (`200`, `404`); POSIX process return values.	few

dition, we are able to eliminate most of the heuristics and guesses for log parsing used by existing solutions.

Insight 2: Common log structures lead to useful features. A person usually reads the log messages in Figure 1 as a constant part (`starting: xact ... is`) and multiple variable parts (`325`/`326` and `COMMITTING`/`ABORTING`). 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 (Lim et al., 2008). 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 (Oliner & Stearley, 2007).

Message variables carry crucial information as well. In contrast to prior work that focuses on numerical variables (Lim et al., 2008; Oliner & Stearley, 2007; Yamanishi & Maruyama, 2005), 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 and state variables.

Identifiers are variables used to identify an object manipulated by the program (e.g., the transaction ids `325` and `346` in Figure 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 1). Table 1 provides additional examples of such variables. We can determine whether a given variable is an identifier or a state variable pragmatically 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; for a detailed discussion see Xu et al. (2009b).

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 for problem detection.

Insight 3: 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 messages by time windows only, and the detection accuracy suffers from noise in the correlation (Jiang et al., 2009; Stearley, 2004; Yamanishi & Maruyama, 2005). 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.

Insight 4: Logs contain strong patterns with lots of noise. Our last but 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. This is because 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 Prin-

cipal 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 introduce our method of combining frequent pattern mining and PCA detection in Section 4.

Case Studies. 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 there are three production systems from real Internet services. Though they are distinct in functionality, developed using different languages by different developers at different times, cover several popular programming languages, including C, C++, Java, Python and ActionScript3, these logs have many common properties. 27 of the 29 systems use free text logs, and our source-code-analysis-based log parsing applies to all of them. In these systems, we found that about 1%-5% of code lines are logging calls in most of the systems. 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 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 the survey (22 have both), confirming our assumption of their prevalence.

3. Methodology Overview

Figure 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).

Our log parsing method consists of two steps: the static source code analysis and the runtime log parsing. The static source analysis step not only extracts all log printing statements from the source code, but also tries to infer types of variables contained in the log messages. Thus we can discover the message format even with the complex type hierarchies in modern object-oriented languages. The runtime log parsing step uses information retrieval techniques to "search" through all possible strings extracted from template for best-matching "schemas" for each log message. The process is stateless, so it is easy to parallelize and implement in a data stream processor in the online setting.

We implemented source code analyzers for Java, C, C++ and Python. In the cases where source code is not readily available or too hard to manage, we can achieve similar results by directly extracting log message template from program binaries. Our experiments show that we can achieve high parsing accuracy in a variety of real-world systems. Xu et al. (2009b) provides more details.

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.

Step 2: Feature creation. Next, we construct feature vectors from the extracted information by choosing appropriate variables and grouping related messages. We focus on two widely applicable features: the message count vectors constructed from identifiers, and the state ratio vectors constructed from state variables[1].

We briefly summarize the construction of message count vectors; readers may refer to Xu et al. (2009b) for further details. We observe that all log messages reporting the same identifier convey a single piece of information about the identifier. By grouping these messages, we get the *message count vector*, which is similar to an execution path (Fonseca et al., 2007). 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 dimension of the vector 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: the "document" is the message group, the dimensions of the vector consist of the union of all useful message types across all

[1] Our method provides flexibility for generating other types of features, including application-specific ones that incorporate operators' domain knowledge.

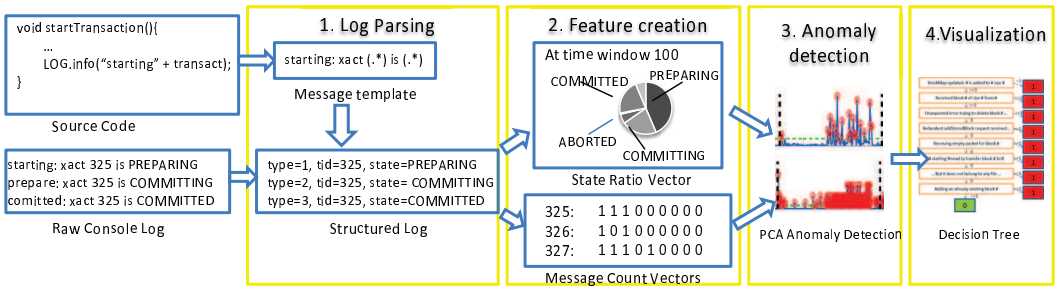

Figure 2. Overview of console log analysis work flow.

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").

Step 3: Machine learning. Next, we apply machine learning methods to analyze feature vectors. In this work, we focus on using anomaly detection techniques to classify each feature vector as normal or abnormal. We find that the Principal Component Analysis (PCA)-based anomaly detection method works very well in our setting (Dunia & Qin, 1997). 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 a frequent-pattern-based method to eliminate the vast majority of normal messages quickly. Combining these two methods, we can achieve an online detection with both small latency and high accuracy.

Notice that the anomaly detection algorithm we chose is not intrinsic to our approach, and different algorithms utilizing different 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* (Witten & Frank, 2000), which provides a more detailed explanation of how the problems are detected, in a form that resembles the event processing rules (Hansen & Atkins, 1993) with which system integrators and operators are familiar. Xu et al. (2009b) provides details about the decision tree constructed.

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.

4. Online Detection

Recall that the message count vectors are created on message groups: We group messages by *identifiers*, and obtain results similar to an event trace of operations on the program object that the identifier represents. Therefore, effective online detection on message count vectors requires striking a balance between accuracy and time to detection (determined only by how long the algorithm has to wait before making a decision). At one extreme, if we wait to see the entire trace before attempting any detection, our results should be as accurate as the offline detection but with excessive time to detection. At the other extreme, if we try to make a decision as soon as a single event appears, we lose the ability to perform anomaly detection based on *sequences* (a group of related events), yet our experience shows that analyzing message sequences instead of the individual events is key to accurate detection, as we discussed in Section 2.

We manage this trade-off by designing a two-stage detection method (Xu et al., 2009a). As depicted in Figure 3, we use a pattern-based filtering to do fast and accurate detection of "normal" events, which consists the majority of all events in logs; for the events that do not match the pattern, we train a PCA model and use the model to decide whether the non-matching sequences are normal.

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 session duration information. As we discussed in Section 2, when a system is under normal operation the majority of operations go through a few normal paths. Using frequent event pat-

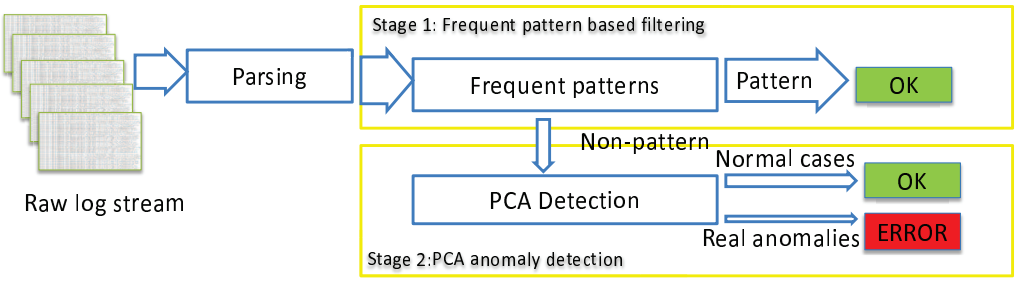

Figure 3. Overview of the two stage online detection systems. The width of arrows qualitatively represents the amount of events going into each stage. The first stage of frequent pattern based filtering is able to handle most normal events.

terns, we can determine as soon as a normal sequence successfully completes; by estimating a duration distribution for each pattern, we can timeout sequences that do not complete in expected latency with high probability.

However, frequent pattern mining is not enough by itself. Random noise (e.g., due to defects in console logs, such as overlapping or incorrect ordering) can be eliminated from patterns only because they do not have enough support. If we reduce the minimal support level, more noise will be introduced to the patterns, reducing the quality of the patterns. To solve the dilemma, we added a second stage, which applies PCA-based anomaly detection to non-pattern events that make it through the first stage.

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 involves domain-specific considerations about console logs.

Our experiments shows that in one of our data sets, the first stage filters over 84% of events, making the detection on these normal events fast. Note also that quick filtering of normal events also saves the amount of memory needed for implementing the detector.

5. Result Highlights

To be succinct yet reveal important issues in console log mining, we focus further discussion on two representative systems: the Darkstar online game server and the Hadoop File System (HDFS). Both of 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. 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. All log data are collected from unmodified off-the-shelf systems "as is," without any modification to the program itself.

Table 2. Data sets used in evaluation. Nodes=Number of nodes in the experiments.

System	Nodes	Messages	Log Size
Darkstar	1	1,640,985	266 MB
Hadoop (HDFS)	203	24,396,061	2412 MB

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 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: excessive transaction aborts and retries when transactions time out due to resource contentions. The cause can only be discovered by analyzing console logs and discovering the patterns of normal states of transactions. Xu et al. (2009b) provides details about the experiment.

Satisfied with Darkstar results and 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 during a two-day period. We successfully extracted log segments indicating runtime performance problems that have been confirmed by Hadoop developers.

Table 3. Precision and recalls of both online and offline detection. Notice that as detailed in Xu et al. (2009a), the online detection precision is lowered by a number of ambiguous cases.

Experiments	Precision	Recall
Online	86%	100%
Offline	91%	99.3%

We achieved highly accurate detection results both with the offline and online algorithms. 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. The detailed evaluation based on Hadoop file system is presented in Xu et al. (2009b) and Xu et al. (2009a).

6. Conclusions and Future Work

We propose a general approach to problem detection via the analysis of console logs, the built-in monitoring information in most software systems. Using source code as a reference to understand the structure of console logs, we are able to parse logs accurately. The accuracy in log parsing allows us to extract the identifiers and state variables, which are widely found in logs yet are usually ignored due to difficulties in log parsing. Using parsed logs, we are able to construct powerful features capturing both global states and individual operation sequences. We eliminated many complexities and forms of noise, which would otherwise present severe challenges to the machine learning algorithms, during feature construction (e.g., by grouping relevant events using identifiers). Thus, simple algorithms such as PCA yield promising anomaly detection results. In order to detect abnormal sequences in an online setting, we adopted a two-stage approach which uses frequent pattern to filter out normal events while using PCA detection to detect the anomalies.

Our work has opened up many new opportunities to turn built-in console logs into a powerful monitoring system for problem detection, and suggests a variety of future directions that can be explored, including: 1) analyze logs from multiple layers or subsystems of the entire application stack in order to understand the interactions among these components; 2) bring the human operators into the debugging loop by allowing operators to give feedback to the detector and incorporate domain knowledge to refine the detection; 3) combine console log information with other structured traces, such as performance counters, resource utilization, etc.; 4) apply more robust learning algorithms to handle defective logs, for example logs containing missing or corrupted messages.

Our current work does not make any change to the log generation code in the program. But we can also aim to improve current console log generation frameworks to allow more dynamic and fine granularity control of individual message types. With such a framework, we can do real-time control of console log generation, which will enable us to further reduce the overhead of generating unnecessary logs, while making sure the interesting and important messages are kept in the logs.

Acknowledgement

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. We also thank Prof. Johannes Fuernkranz and the program committee for inviting us to present this work at ICML.

References

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

Chen, M. Y. and Brewer, E. A. Path-based failure and evolution management. In *Proceedings of NSDI'04*, San Francisco, CA, 2004.

Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A., and Brewer, E. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of DSN'02*, Washington, DC, 2002.

Cohen, I., Zhang, S., Goldszmidt, M., Symons, J., Kelly, T., and Fox, A. Capturing, indexing, clustering, and retrieving system history. In *Proceedings of ACM SOSP'05*, Brighton, UK, 2005.

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

Fisher, K., Walker, D., Zhu, K. Q., and White, P. From dirt to shovels: fully automatic tool generation from ad hoc data. In *Proceedings of POPL'08*, San Francisco, CA, 2008.

Fonseca, R., Porter, G., Katz, R. H., Shenker, S., and Stoica, I. Xtrace: A pervasive network tracing framework. In *In Prococeedings of NSDI'07*, Cambridge, MA, 2007.

Ganapathi, A., Kuno, H., Dayal, U., Wiener, J. L., Fox, A., Jordan, M., and Patterson, D. Predicting multiple performance metrics for queries: Better decisions enabled by machine learning. In *Proceedings of ICDE'09*, Shanghai, China, 2009.

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

Hansen, S. E. and Atkins, E. T. Automated system monitoring and notification with Swatch. In *Proceedings of LISA'93*, Monterey, CA, 1993.

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

Jiang, W., Hu, C., Pasupathy, S., Kanevsky, A., Li, Z., and Zhou, Y. Understanding customer problem troubleshooting from storage system logs. In *Proccedings of FAST'09*, San Francisco, California, 2009.

Karagiannis, T., Papagiannaki, K., Taft, N., and Faloutsos, M. Profiling the end hos. In *Proceedings of Passive and Active Measurement Workshop (PAM'07)*, Belgium, 2007.

Lakhina, A., Crovella, M., and Diot, C. Diagnosing network-wide traffic anomalies. In *Proceedings of ACM SIGCOMM*, Portland, OR, 2004.

Li, Z., Lu, S., Myagmar, S., and Zhou, Y. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of OSDI'04*, San Francisco, CA, 2004.

Lim, C., Singh, N., and Yajnik, S. A log mining approach to failure analysis of enterprise telephony systems. In *Proceedings of DSN'08*, June 2008.

Ma, S. and Hellerstein, J. L. Mining partially periodic event patterns with unknown periods. In *Proceedings of ICDE'01*, Washington, DC, 2001.

Makanju, A. A.O., Zincir-Heywood, A. N., and Milios, E. E. Clustering event logs using iterative partitioning. In *Proceedings of KDD '09*, Paris, France, 2009.

Oliner, A. and Stearley, J. What supercomputers say: A study of five system logs. In *Proceedings of DSN'07*, 2007.

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

Splunk. *Splunk user guide*. Splunk Inc., Sept 2008.

Stearley, J. Towards informatic analysis of syslogs. In *Proceedings of IEEE CLUSTER*, Washington, DC, 2004.

Vaarandi, R. A data clustering algorithm for mining patterns from event logs. *Proceedings of IEEE IPOM'03*, 2003.

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

Wang, H. J., Platt, J. C., Chen, Y., Zhang, R., and Wang, Y. Automatic misconfiguration troubleshooting with peerpressure. In *Proceedings of OSDI'04*, San Francisco, CA, 2004.

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

Xu, K., Zhang, Z., and Bhattacharyya, S. Profiling internet backbonetraffic: Behavior models and applications. In *Proceedings of ACM SIGCOMM*, 2005.

Xu, W., Huang, L., Fox, A., Patterson, D., and Jordan, M. Online system problem detection by mining patterns of console logs. In *Proceedings of ICDM'09*, Miami, FL, 2009a.

Xu, W., Huang, L., Fox, A., Patterson, D., and Jordan, M. Large-scale system problems detection by mining console logs. In *Proceedings of SOSP'09*, Big Sky, MT, 2009b.

Yamanishi, K. and Maruyama, Y. Dynamic syslog mining for network failure monitoring. In *Proceedings of ACM KDD'05*, Chicago, IL, 2005.

Ye, Y., Wang, D., Li, T., and Ye, D. IMDS: Intelligent malware detection system. In *Proceedings of ACM KDD'07*, 2007.

Zheng, A. X., Jordan, M. I., Liblit, B., Naik, M., and Aiken, A. Statistical debugging: Simultaneous isolation of multiple bugs. In *Proceedings of ICML'06*, 2006.