Operational Log Analysis for Big Data Systems: Challenges and Solutions

Andriy Miranskyy, *Member, IEEE,* Abdelwahab Hamou-Lhadj, *Member, IEEE*, Enzo Cialini, and Alf Larsson

***Abstract*— Big Data Systems (BDS) are known to be complex, consisting of multiple interacting hardware and software components, such as distributed compute nodes, databases, middleware, etc. Any of these components can fail. Finding the root causes of such failures is an extremely laborious process. An analysis of logs, generated by a BDS during its operation, can speed up this process. The logs can also be used to improve testing processes, detect security breaches, customize operational profiles, and any other tasks that require the analysis of run-time data. However, the adoption of log analysis tools is hampered by practical challenges. The logs emitted by a BDS can be thought of as Big Data themselves. In this paper, we report on the major issue areas commonly faced by practitioners when working with large logs. We also propose practical solutions, while highlighting the remaining challenges.**

***Index Terms*—Big Data Systems, Software Tracing and Logging,**

I. Introduction

Big Data Systems (BDS) are complex and have many dynamic components including distributed computing nodes, networking, databases, middleware, a Business Intelligence (BI) layer, High Availability infrastructure, etc. Any of the components (and their interactions with others) can fail, leading to a crash of the system or quality degradation (e.g., performance, reliability, security). Finding a root cause of these problems is non-trivial, because BDS components depend on each other. For example, a database's failure to access data might be caused not by a defect in the database, but by corruption in the underlying distributed storage systems.

Typically, an analyst resorts to examining operational data, namely logs and traces, generated by the BDS components, trying to pinpoint the root cause of the problem. A log or a trace is a sequence of temporal events captured during a particular execution of a system. For example, a log can contain software execution paths, events triggered during software execution, or user activities. There is no clear distinction between logs and traces. Often the term log is used to represent the way a program is used (such as security logs), while tracing is used to capture a program's elements that are invoked in a given execution of the system. Tracing is used for debugging and program understanding. In this paper, we use the terms logs and traces interchangeably.

A. Miranskyy is with the Department of Computer Science, Ryerson, University, Toronto, ON, Canada (e-mail: avm@ryerson.ca).

A. Hamou-Lhadj is with the Department of Electrical and Computer Engineering, Concordia University, Montreal, QC, Canada (e-mail: wahab.hamou-lhadj@concordia.ca).

E. Cialini is with IBM, Toronto, ON, Canada (e-mail: ecialini@ca.ibm.com).

Alf Larsson is with Ericsson, Stockholm, Sweden (e-mail: alf.larsson@ericsson.com).

Logs have been shown to be essential in several software engineering tasks including debugging [1], defect analysis log [2]–[4], testing [5], detecting security breaches[6], and customizing operational profiles [7]. A good overview of the application domains of log analysis can be found in [8]. The authors discuss the applications of log analysis to performance analysis, security, prediction, and profiling.

Independent of the data captured by the log and the log's area of usage, there is a number of characteristics that all of these logs share. These characteristics make it difficult to work with logs in industrial settings. Peculiarly, the same characteristics are used to describe the properties of Big Data. The characteristics are as follows.

- Velocity: the data (in some cases) must be processed in real time.
- Volume: mountain ranges of historical data.
- Variety: captured data can be structured or unstructured.
- Veracity: captured data must be cleaned.
- Value: not all captured data are useful.

Essentially, BDSs designed to process Big Data usually emit Big Data (captured in logs) themselves [9]. Of course, not all BDSs generate large volumes of logs. Also, small systems may generate a large amount of data. However, for most BDS-emitted logs, an analyst will observe at least one of the Big Data characteristics.

The fundamental processes in leveraging log data include building solutions for delivering, storing, and "crunching" large volumes of data. Each of these processes comes with a myriad of challenges. In this paper, we draw on our experience working on analyzing large logs at both IBM and Ericsson in the context of industrial projects. More particularly, we discuss seven issues that practitioners in both companies constantly face when working with large logs: namely, storing logs, scalable analysis of log data, accurate capturing and replaying of logs, inadequate tooling for processing logs, and problems with classifying and formatting logs. We describe these issues by mapping to those commonly found in analyzing big data. We also discuss possible solutions.

II. Log Analysis: Challenges

A. Scarce Storage

In this section, we describe issues that arise due to a large *volume* of logs that must be stored and compared. The first issue arises while uploading a log to a storage facility for processing. A log, even compressed by a mainstream archival utility, such as zip, can reach tens of gigabytes. If the log is collected in-house, copying the file from the machine where the log was collected to the storage facility is a fast and straightforward task, as internal networks are typically fast. However, if the file is collected at a remote location, e.g., a customer site, the process becomes challenging due to network bandwidth caps and fragile connections. For example, a 50GB file transfer on a 1Gbit network, at best, will take approximately 7 minutes, while it will take approximately 12 hours on a 10Mbit network, and 5 days on a 1Mbit network. Note that multiple files may be uploaded simultaneously, further increasing the upload time.

If the support team needs the file urgently (e.g., to diagnose a crash of a production BDS), shipping the log file via a courier on a physical storage device may be considered. Another option is to give the support team remote access to the customer site (if customer's security policy permits this) to work with the file manually. However, this typically implies that the file must be processed manually; therefore, it does not speed up the automatic diagnostics.

The second issue is related to the growth of the log repository. The number of logs grows rapidly as time passes. To illustrate this, consider the following two real-world cases: In one case, a company is gathering logs from clients to automatically detect rediscovered (i.e., recurring) defects, speeding up problem diagnostics. They collect 20,000 logs per year, ranging in size from 1KB to 100GB. The logs contain software execution data ranging from stack dump to full execution paths with parameter values. Their repository grows at a rate of 0.5PB a year. In another case, an advertisement company collects bidding logs on advertisement banners to detect fraudulent activity (robotic clicks). They track information from 1.5 billion requests (bids) per day, collecting 1.5PB of bidding logs per year.

One possible solution to this problem is to distribute large logs on various storage devices. Storing large volumes of data is expensive, and multiple approaches to designing storage solutions exist. Ideally, all data should be kept in a repository where accessing the data can be done instantaneously, such as in in-memory (cache) databases. Unfortunately, large volumes of log data make this approach prohibitively expensive. To find a compromise between efficiency and cost, one puts 1) frequently accessed data on fast but expensive storage devices and 2) infrequently accessed data on slow but inexpensive storage devices.

Another possibility is to design a storage solution where the amount of logs that must be stored can also be reduced. In the simplest form, logs older than a certain time threshold, e.g., three years, can be purged. However, this approach must be used carefully, as customers often rediscover old problems because many install fix-packs reluctantly. For example, we have seen defects being rediscovered by clients three years after a fix-pack, with the patch for the defect, was made available.

We can also eliminate parts of logs that are not useful for analysis instead of eliminating a complete log. For example, if an execution log is used to find rediscovered (i.e., recurring) defects [10], a part of the execution path that represents the defect-specific code path may be kept and the rest may be eliminated. In some cases, this can be done online while the log is being processed for the first time. Offline log filtering is also possible, but this requires saving the original logs and expecting the users to eliminate undesirable parts. Log abstraction techniques exist (e.g., [11], [12]), allowing users to automatically reduce the log size, while keeping as much of the essence as possible (see Section II.F for details). Though they vary in design, most existing approaches focus on eliminating low-level implementation details, not always required to understand the behaviour of a complex scenario. How log abstraction can be used to solve specific maintenance tasks, such as defect discovery and bug fixing, is still unclear.

Finally, sampling techniques have also been used to reduce the size of logs [13]. Sampling consists of selecting parts of a log instead of analyzing the entire content. The problem with sampling is that the resulting log may not contain all the information needed for analysis (e.g., rare events [8]). Moreover, many sampling approaches need manual tuning of parameters. Finding the right parameters can be a difficult task: if some parameters work well for one system, they might not work for another one.

B. Unscalable Log Analysis

As mentioned above, detailed logs generated by a busy BDS can easily reach tens of terabytes. It can be attested, based on the authors' practical experience, that crawling through such logs is laborious and expensive. For example, manual determination of a fault's root cause can consume 30– 40% of the total time needed to fix a problem [2]. Therefore, techniques must be leveraged from the domain of operational data/dynamic analysis, which can process large *volumes* of logs emitted by BDS components. Moreover, in some cases (e.g., to detect fraud or security threats), data emitted by BDSs must be processed in real-time, making *velocity* an important characteristic.

Lossless log analysis techniques (e.g., representing a log as a Finite State Automata [3] or Signal [14]) are accurate but not scalable [15] (as they must deal with large volumes of uncompressed logs); lossy techniques are scalable, but not universal [15].

For example, a single log must be compared against a library of reference logs, say, to identify a recurring defect. If the library contains 1 PB of logs, simply reading these logs into memory for the purpose of comparison will take a significant amount of time (even in parallel on multiple computers).

To accelerate the comparisons, an iterative approach must be used, such as the scalable iterative-unfolding technique (SIFT) [15]. The logs are first compacted using various lossy compression techniques: the higher the compression, the smaller the amount of information remaining and the faster the comparison. Then, logs are iteratively compared at different levels of compression, from high (where processing is fastest) to low (where processing is slowest).

The process rapidly eliminates dissimilar logs, eventually leaving residual, similar logs at the lowest level of compaction. Identified similar logs can be passed to external tools for further analysis. Typically, logs would be stored at the highest level of compaction in hot storage, at intermediate levels in warm storage, and at the lowest levels in cold storage.

Comparison techniques, such as SIFT, can be parallelized. A single log comparison against a library of logs, e.g., for defect detection [15], is an "embarrassingly parallel" task (because comparisons of each pair of logs are independent of each other). Therefore, the comparison can be easily parallelized using the MapReduce programing model (e.g., using the Apache Hadoop platform). If interactions between comparison processes are required, e.g., for clustering logs to improve testing [15], the Apache Spark (or a similar) platform is better suited.

The existing commercial solutions, such as Apache Chukwa, HP IT Operations Analytics, IBM Operations Analytics – Log Analysis, and Splunk products, use MapReduce to analyze and visualize different types of log data. They take logs from various sources as input data and index them as a structural schema data. Then, the query-like programming which is similar to SQL is performed on the structural schema data.

C. *Inaccurate Capture-Replay*

This section discusses the accurate capturing of logs on a production system and replaying/aligning them on a test system to ensure accurate testing and diagnostics. BDS components talk to each other, with their subcomponents distributed through a cluster of computers. Additionally, each BDS component may have multiple processes and threads running in parallel, adding to the complexity. Large *volumes* of logs on a busy BDS are generated with high *velocity*.

The larger the volume and velocity, the higher the contribution of the observer effect, which describes a phenomenon of changing the system while measuring its attributes. For example, when tire pressure is measured using a tire pressure gauge, some air escapes from the tire, changing the tire pressure.

In BDS systems, enabling tracing mechanisms leads to a system slowdown, as extra resources must be allocated to capture and store the log. The higher the intensity of the workload, the higher the observer effect, as more resources will be needed to capture the activities. This becomes especially important when trying to capture data for a heisenbug, e.g., a timing-related one: when the system slows down, the timing problems may disappear, as the chance of race conditions, deadlocks, etc. will decrease.

Thus, it is important to build capturing infrastructure that will minimize the observer effect. Essentially, a tracing infrastructure should not slow down the BDS significantly.

Software- and hardware-based solutions exist, many of which are platform-specific. Typically, software systems are more prone to the observer effect, but are more universal. Hardware systems, on the other hand, tend to be less intrusive, but are very platform-specific. Let us look at some of the readily available tooling.

Software-based logging solutions can be grouped into four categories:

- Integrated into an Operating System (OS);
- Compiler-based tooling;
- Custom build loggers;
- Specialized solutions.

OS-level tooling appears in multiple OSs. For example, a framework called DTrace exists, which, once enabled, captures the execution path of a program in real time. The developer need not make any modifications to the code to enable this instrumentation. It is available as part of the BSD kernel, making the tool available on Solaris, Mac OS X, FreeBSD, and NetBSD OSs. An unofficial port also exists from DTrace to Linux kernel.

Compiler-based tools exist that capture information about code execution. For example, Intel Compiler Code Coverage or GNU gcov can capture information about executed code blocks. The code must be recompiled to enable this tooling. The overhead associated with the tooling is low, but captured data are limited, as information about the sequence in which code blocks are executed and about what data is passed from code block to code block is lost.

On the other side of the spectrum, tools, such as Intel Parallel Studio (IPS), exist to capture information from multi-threaded programs, track the state of shared memory, etc. The tool is extremely useful for capturing and diagnosing problems in multi-process and thread environments. Unfortunately, the observer effect of the IPS is very pronounced, as performance degradation can reach multiple magnitudes.

Custom-built solutions, by construction, vary widely. In this case, developers build logging infrastructure from scratch or re-use language-specific logging libraries (such as Apache log4j). Hence, a developer must instrument the code with probe points manually, specifying information that must be captured at every probe point. Typically, probe points are located near the entries and exits to the functions and near important branching points.

Specialized solutions are capable of capturing specialized types of logs and are less universal than the above-described categories of logs. For example, in the database area, tools exist (such as IBM InfoSphere Optim Workload Replay and Oracle Database Replay) for capturing workloads on a production system and replaying them on a test system to ensure accurate system testing. The tools work with minimal intrusion and slowdown and they can often be configured so the information about the workload is read directly off network cards. However, the tooling will not capture low-level information about what is happening in the database engine.

Hardware-based logging can also be used. In this case, the information about execution logs is captured at the hardware level, minimizing the observer effect; in addition, information is available at a very low level (often at the level of CPU instructions). Currently, these solutions are limited to few platforms. For example, IBM Mainframe's z/OS can capture system- and transaction-level logs. Intel has been working on building processor-level tracing into their products [16], but no commercial offering exists at this stage.

Regarding log replay, there are currently more questions than answers. As discussed above, specialized tools in the database area allow both workload capturing and replaying. However, they focus on relational databases. There is a need for tools that can capture intensive production workloads of BDSs and replay them on the test system in the presence of data obfuscation. Likewise, tools for other components of BDSs, such as a BI layer, are required. There is also a need for a general strategy for scaling down the workload. If, say, the test system is 10 times smaller than the production system, does this mean (in terms of: number of concurrent connections, operations per unit of time, etc.) the workload should be reduced by 10 times?

D. Inadequate Tooling for Instrumenting BDS Source Code

The volume problem manifests itself not only in the large volume of data generated by the BDS itself, but also in the large *volume* of BDS components' source codes that must be instrumented.

Enterprise-level software consists of millions of lines of source code; not every tool is capable of handling such volumes. Typically, this will be manifested by a crash of the instrumented code, a return of incorrect results, dramatic performance degradation, etc. These symptoms can be caused by various factors, e.g., the observer effect (tooling interferes with normal component execution), scalability (overflows in internal tooling data structures), and incorrect code instrumentation (due to issues with tooling's code parsers).

Tool vendors are typically open to fixing the problems. Unfortunately, BDS component developers may be unable to share their source code due to non-disclosure agreements, In this case BDS developers have to build self-contained test cases that can be passed to the tool developers in question, which is laborious.

Performance degradation caused by the log capturing tools may exacerbate the observer effect (discussed in the previous section). In addition, customers may not permit the use of such slow tools with their production systems.

Even in the case of in-house test systems, performance may be important. For example, we may run nightly regression tests in parallel on 100 computers to complete test executions by morning. Capturing execution logs from the test executions helps to diagnose automatically the root causes of test failures. However, even a 50% performance degradation (due to code instrumentation) would require 50 additional computers for regression tests runs, increasing test cost significantly.

E. Incorrect Log Classification

Once the logs to the central repository are captured and uploaded, they must be cleaned due to the *veracity* of the logs being captured. It must also be ensured that only required logs are kept for future analysis and classification, as not all logs have *value*.

Consider the following use case. A log was collected to diagnose the problem, but whether the log captured the root cause of the problem is uncertain. A number of reasons exist for why the problem cannot be captured reliably. In the simplest case, when the software crashes and a stack dump is gathered, the root cause can typically be captured on the first try (unless we have an extremely pathological case leading to stack corruption). However, if we are trying to capture a root cause of an intermittent defect, a test case may need to be run multiple times to capture the problem. For example, the problem can be caused by a heisenbug and because "stars do not align" to trigger it the first time. Alternatively, for example, the test case may require running multiple times, because log-capturing tooling for some components of the BDS must be disabled (to minimize performance degradation); by Murphy's Law, these were the components required for problem diagnostics.

Unfortunately, the logs from all the tries will often be loaded to the storage facility. This makes supervised classification problematic, as which logs contain defect patterns (and have value) and which do not (and are worthless) are often unknown. Without knowing this information, the classification models' accuracy reduces significantly, as all logs associated with a given problem must be classified as logs that contain information about root causes (even though this is not always the case), confusing the classifier.

In this case, manual intervention is often required. Developers need to eliminate manually the worthless logs. A person responsible for maintaining the data repository in a clean and consistent state must be selected. This person must follow up with developers and testers, ensuring that only valuable logs are kept.

F. Variety of Log Formats

As discussed in Sections I and II.C, a BDS consists of multiple software and hardware components. Even though universal formats exist [11], [17], [18], they are not widely used. Therefore, these components will emit heterogeneous logs in a *variety* of formats.

To perform the analysis, all logs need to be converted to one unified representation. Typically, log analysis tool developers resort to building converters for each data format. However, no universal converter exists.

In addition, current universal formats have limitations. For example, the Knowledge Discovery Metamodel (KDM) [17] is an Object Management Group standard that supports tracing but does not support any compaction mechanism. In other words, a log will be represented in its original format, which hinders scalability. The Compact Trace Format (CTF) [12] is an example of a metamodel that models log information in a compact way (compacted logs can be compared without being "uncompacted"). It is however limited to simple function call traces. The CTF uses a graph theory concept to turn a routine call tree into an ordered directed acyclic (DAG) graph by representing similar sub-trees only once. This way, a log should never be saved as a tree but rather as a DAG. The authors of the CTF showed this compaction technique scales well to large logs.

However, although the CTF has been extended to support multi-process systems [12], it is still not expressive enough to be widely deployed. For example, it lacks supports for function arguments, statements, etc.

Creating a universal log format to support all kinds of logs is challenging and even if successful, this format may end up being ineffective. This is because the need for a log type depends on the application domain. Take, for example, logs of system calls. These are used extensively in security (anomaly detection in particular) and in understanding how applications interact with the OS. These logs may be of little use for analysts who wish to understand the application design. Perhaps the objective should be to create standardized formats for different application domains yet to be defined.

G. Inadequate Privacy of Sensitive Data

Due to space constraint, we briefly highlight some key issues that relate to the *veracity* of sensitive data captured in a log, e.g., usernames, passwords, and credit card numbers.

The first issue is related to system debugging. One can use logs from production system (see [19]) to reproduce a bug exposed on production system in a lab environment, as discussed in Section II.C. Existing bug reproduction techniques rely on instrumenting the system in order to capture objects and other system components at run-time. When a faulty behaviour occurs in the field, the stored objects and often the entire memory dump are sent to the developers to reproduce the crash. Unfortunately, the collected objects may contain sensitive information causing customer privacy issues.

One solution to this issue is using obfuscation and anonymization techniques. Some logging frameworks have rule-based extensions that obfuscate or remove sensitive data before it is stored in the log. However, such an approach is not entirely error-proof, because it depends on the accuracy of detection rules, which often change as the BDS source code evolves. Potentially, in the future, a tool will be created that would automatically and accurately recognize sensitive data and obfuscate it while in progress.

Another issue comes from the fact that due to legal constraints, a user may be unable to share the log with a BDS manufacturer (this is the case with, e.g., financial or military organizations). There are two solutions to this issue.

The first is to provide a standalone version of a log processing and analysis tool to the client. This tool should be able to analyze the trace/log remotely, without requiring any external information. The feasibility of this solution would depend on the amount of storage and computing power required by the tool.

The second solution is to encrypt the log using homomorphic encryption, so the BDS manufacturer can perform the analysis without being able to extract sensitive information from the log. However, this solution is very computationally expensive and is not practical at this stage.

III. Summary

In this paper, seven areas of issues that practitioners face while building solutions for storing and processing the logs were discussed. The root causes of the issues lie in the characteristics of the logs. We highlighted existing solutions to the issues and posed unanswered questions, based on the authors' experience as well as the interactions they had with industrial partners building such solutions.

We believe that issues and solutions discussed in this paper are of interest to practitioners, as they can readily leverage existing techniques to build their own solutions. The findings are also of interest to the academic community, as they highlight practical problems that remain unsolved.

References

[1] A.Oliner and J.Stearley, "What Supercomputers Say: A Study of Five System Logs," in *International Conference on Dependable Systems and Networks, 2007. DSN '07*, 2007, pp. 575–584.

[2] S.S.Murtaza, et al., "An empirical study on the use of mutant traces for diagnosis of faults in deployed systems," *J. Syst. Softw.*, vol. 90, pp. 29–44, 2014.

[3] L.Mariani, F.Pastore, M.Pezze, "Dynamic Analysis for Diagnosing Integration Faults," *IEEE Trans. Softw. Eng.*, vol. 37, no. 4, pp. 486–508, 2011.

[4] R.P.J.C.Bose and W.M.P. van der Aalst, "Discovering signature patterns from event logs," in *IEEE Symposium on Computational Intelligence and Data Mining*, 2013, pp. 111–118.

[5] D.Cotroneo et al., "Investigation of Failure Causes in Workload-driven Reliability Testing," in *Fourth International Workshop on Software Quality Assurance*, 2007, pp. 78–85.

[6] W.Lee, S.J.Stolfo, P.K.Chan, "Learning Patterns from Unix Process Execution Traces for Intrusion Detection," *AAAI Workshop AI Approaches Fraud Detect. Risk Manag.*, pp. 50–56, 1997.

[7] A.E.Hassan et al., "An Industrial Case Study of Customizing Operational Profiles Using Log Compression," in *International Conference on Software Engineering*, 2008, pp. 713–723.

[8] A.Oliner, A.Ganapathi, and W.Xu, "Advances and Challenges in Log Analysis," *Commun ACM*, vol. 55, no. 2, pp. 55–61, 2012.

[9] A.Mockus, "Engineering Big Data Solutions," in *Future of Software Engineering*, 2014, pp. 85–99.

[10] T.Reidemeister et al., "Diagnosis of Recurrent Faults Using Log Files," in *Conference of the Center for Advanced Studies on Collaborative Research*, pp. 12–23.

[11] R.Brown et al., "STEP: A Framework for the Efficient Encoding of General Trace Data," in *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2002, pp. 27–34.

[12] A.Hamou-Lhadj and T.C.Lethbridge, "A Metamodel for the Compact but Lossless Exchange of Execution Traces," *Softw Syst Model*, vol. 11, no. 1, pp. 77–98, 2012.

[13] H.Pirzadeh et al., "Stratified Sampling of Execution Traces: Execution Phases Serving As Strata," *Sci Comput Program*, vol. 78, no. 8, pp. 1099–1118, 2013.

[14] A.Kuhn and O.Greevy, “Exploiting the Analogy Between Traces and Signal Processing,” in *IEEE International Conference on Software Maintenance*, , 2006, pp. 320–329.

[15] A. V. Miranskyy et al., “SIFT: a scalable iterative-unfolding technique for filtering execution traces,” in *Conference of the center for advanced studies on collaborative research*, 2008, pp. 21:274–21:288.

[16] “Processor Tracing | Intel® Developer Zone.”, https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing. [Accessed: 07-Aug-2015].

[17] “Information technology - Architecture-Driven Modernization (ADM): Knowledge Discovery Meta-Model (KDM),” Object Management Group, ISO/IEC 19506:2012(E), Apr. 2012.

[18] G. Lee et al., “The Unified Logging Infrastructure for Data Analytics at Twitter,” *in VLDB Endow*, vol. 5, no. 12, pp. 1771–1780, 2012.

[19] H. Jaygarl et al., “OCAT: Object Capture-based Automated Testing,” in *International Symposium on Software Testing and Analysis*, 2010, pp. 159–170.