Operational-Log Analysis for Big Data Systems

Challenges and Solutions

Andriy Miranskyy, Ryerson University

Abdelwahab Hamou-Lhadj, Concordia University, Montreal

Enzo Cialini, IBM

Alf Larsson, Ericsson

// When working with large logs, practitioners often face issues such as scarce storage, unscalable analysis tools, innacurate capture and replay of logs, and inadequate privacy. Researchers have devised some practical solutions, but important challenges remain. //

BIG DATA SYSTEMS (BDSs) are complex and have many dynamic components, including distributed computing nodes, networks, databases, middleware, a business intelligence layer, and high-availability infrastructure. Any component (and its interactions with others) can fail, leading to a system crash or degraded quality (for example, performance, reliability, or security). Finding these problems' root cause is nontrivial because BDS components are interdependent. 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.

To pinpoint a problem's root cause, analysts typically examine operational data—logs and traces—generated by the BDS components. A log or 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. No clear distinction exists between logs and traces. Often, the term "log" represents how a program is used (such as security logs), whereas "tracing" captures a program's elements that are invoked in a given execution of the system. Tracing is used for debugging and program understanding. In this article, we primarily use the term "log." (For more on data logging, see the sidebar.)

Logs share several characteristics that make working with them difficult in industrial settings:

- *Velocity*. The data (in some cases) requires real-time processing.
- *Volume*. Logs can contain huge amounts of historical data.
- *Variety*. The captured data can be structured or unstructured.
- *Veracity*. The captured data requires cleaning.
- *Value*. Not all the captured data is useful.

These characteristics also describe big data.

Essentially, BDSs designed to process big data usually emit big data (captured in logs) themselves.[1] Of course, not all BDSs generate large volumes of logs. Also, small systems might generate big data. However, most BDS-emitted logs will exhibit at least one big data characteristic.

To leverage log data, developers need ways to effectively deliver, store, and crunch large volumes of data. Each of these processes poses challenges. When analyzing large logs for industrial projects at IBM and Ericsson, we've run into seven issues:

- scarce storage,
- unscalable log analysis,
- inaccurate capture and replay of logs,
- inadequate log-processing tools,
- incorrect log classification,
- a variety of log formats, and
- inadequate privacy of sensitive data.

Here, we map these issues to the related issues in big data analysis and discuss possible solutions.

RELATED WORK IN DATA LOGGING

Logs are essential in software engineering tasks including debugging,[1] analyzing defects,[2–4] testing,[5] detecting security breaches,[6] and customizing operational profiles.[7] Adam Oliner and his colleagues provided a good overview of log analysis application domains; they've applied log analysis to performance analysis, security, prediction, and profiling.[8]

References

1. A. Oliner and J. Stearley, "What Supercomputers Say: A Study of Five System Logs," *Proc. 2007 Int'l Conf. Dependable Systems and Networks* (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. Systems and Software*, Apr. 2014, pp. 29–44.
3. L. Mariani, F. Pastore, and M. Pezze, "Dynamic Analysis for Diagnosing Integration Faults," *IEEE Trans. Software Eng.*, vol. 37, no. 4, 2011, pp. 486–508.
4. R.P.J.C. Bose and W.M.P. van der Aalst, "Discovering Signature Patterns from Event Logs," *Proc. 2013 IEEE Symp. Computational Intelligence and Data Mining*, 2013, pp. 111–118.
5. D. Cotroneo et al., "Investigation of Failure Causes in Workload-Driven Reliability Testing," *Proc. 4th Int'l Workshop Software Quality Assurance*, 2007, pp. 78–85.
6. W. Lee, S.J. Stolfo, and P.K. Chan, "Learning Patterns from Unix Process Execution Traces for Intrusion Detection," *Proc. AAAI Workshop AI Approaches to Fraud Detection and Risk Management*, 1997, pp. 50–56.
7. A.E. Hassan et al., "An Industrial Case Study of Customizing Operational Profiles Using Log Compression," *Proc. 2008 Int'l Conf. Software Eng.* (ICSE 08), 2008, pp. 713–723.
8. A. Oliner, A. Ganapathi, and W. Xu, "Advances and Challenges in Log Analysis," *Comm. ACM*, vol. 55, no. 2, 2012, pp. 55–61.

Scarce Storage

Issues arise when you must store and compare a large volume of logs. One issue arises while you're uploading a log to a remote storage facility for processing. Performing the analysis onsite is usually challenging because of the lack of resources and tools that can diagnose the problem's cause onsite. 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 on which the log was collected to the storage facility is fast and straightforward because internal networks are typically fast.

However, if the file is collected at a remote location—for example, a customer site—the process becomes challenging owing to network bandwidth caps and fragile connections. The number of network nodes and the frequency of the log data create congestion. For example, a 50-Gbyte file transfer takes approximately 7 minutes on a 1-Gbit network but takes approximately 12 hours on a 10-Mbit network and 5 days on a 1-Mbit network. Multiple files might be uploaded simultaneously, further increasing the upload time.

If the support team urgently needs the file (for example, to diagnose a production BDS crash), it could ship the log file on a physical storage device via courier. Another option is to send field engineers to the site to correct the problem. These engineers, however, will most likely need access to the source code and debugging tools. A practical solution is to give the support team remote access to the customer site (if the customer's security policy permits this) to work with the file manually. However, this typically implies that the file must be processed manually, which won't speed up the automatic diagnostics.

Another issue concerns the log repository's growth. The number of logs grows rapidly over time. Consider two real-world cases. In the first case, a company gathers logs from clients to automatically detect rediscovered (recurring) defects, speeding up problem diagnostics. It collects 20,000 logs per year, ranging from 1 Kbyte to 100 Gbytes. The logs contain software execution data ranging from stack dumps to full execution paths with parameter values. The repository grows at 0.5 Pbytes a year. In the second case, an advertising company collects bidding logs on banner ads to detect fraudulent activity (robotic clicks). It tracks information from 1.5 billion requests (bids) per day, collecting 1.5 Pbytes 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, but many storage solutions exist. Ideally, all data should be in a repository that allows instantaneous data access, such as an in-memory (cache) database. Unfortunately, large volumes of log data make this approach prohibitively expensive. To compromise between efficiency and cost, you can put frequently accessed data on fast but expensive storage devices and infrequently accessed data on slow but inexpensive storage devices.

Another possibility is a storage solution in which you reduce the number of logs that must be stored. In the simplest form, logs older than a certain time threshold—for example, three years—can be purged. However, you must use this approach carefully; many customers often rediscover old problems because they're reluctant to install fix packs. For example, we've seen clients rediscover defects three years after a fix pack with the patch for those defects became available.

You can also eliminate parts of logs that aren't useful for analysis instead of eliminating a complete log. For example, if you use an execution log to find rediscovered defects,[2] you can keep the part of the execution path that represents the defect-specific code path and eliminate the rest. In some cases, you can do this 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 lets users automatically reduce the log size while keeping as much of the essence as possible (we further discuss this later).[3,4] Most of these approaches eliminate low-level implementation details that aren't always required to understand a complex scenario's behavior. How to use log abstraction to solve specific maintenance tasks, such as defect discovery and bug fixing, is still unclear.

Finally, sampling techniques can reduce log size by selecting parts of a log instead of analyzing the entire content.[5] However, the resulting log might not contain all the information needed for analysis (for example, rare events[6]) unless sampling is carefully performed taking into account domain knowledge. Moreover, many sampling techniques need manual parameter tuning. Finding the right parameters can be difficult: parameters that work well for one system might not work for another one.

To accelerate the comparisons, an iterative approach is necessary, such as the scalable iterative-unfolding technique (SIFT).

Unscalable Log Analysis

As we mentioned before, logs can easily reach tens of terabytes. As we know from experience, crawling through such logs is laborious and expensive. For example, manual determination of a fault's root cause can consume 30 to 40 percent of the total time needed to fix a problem.[7] So, developers must leverage techniques from operational data analysis or dynamic analysis that can process the large volumes of logs emitted by BDS components. Moreover, in some cases (for example, to detect fraud or security threats), BDS data requires real-time processing, making velocity important.

Lossless log analysis techniques (for example, representing logs as finite-state automata[8] or signals[9]) are accurate but not scalable.[10] (This is because they must deal with large volumes of uncompressed logs.) Lossy techniques are scalable but not universal.[10]

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

To accelerate the comparisons, an iterative approach is necessary, such as the scalable iterative-unfolding technique (SIFT).[10] SIFT first compacts the logs using various lossy-compression techniques: the higher the compression, the less information remaining and the faster the comparison. Then, SIFT iteratively compares the logs at different compression levels, from high (where processing is fastest) to low (where processing is slowest). The process rapidly eliminates dissimilar logs, leaving residual, similar logs at the lowest compression level. You can pass those similar logs to external tools for further analysis.

Typically, logs are stored at the highest level of compression in hot storage, at intermediate levels in warm storage, and at the lowest levels in cold storage.

Comparison techniques such as SIFT are parallelizable. Comparing a log against a library of logs is an embarrassingly parallel task (because comparisons of each pair of logs are independent of each other). So, you can easily parallelize the comparison using the MapReduce programing model (for example, using Apache Hadoop). If you require interactions between comparison processes—for example, for clustering logs to improve testing[10]—Apache Spark (or a similar platform) is better suited. You can also compare logs by grouping them into high-level abstractions (for example, hardware or software events, alarms, and resource overloads). However, this approach might require extensive domain knowledge to guide the abstraction process.

Commercial solutions, such as Apache Chukwa, HP Operations Analytics, IBM Operations Analytics—Log Analysis, and Splunk's products, use MapReduce to analyze and visualize types of log data. They take logs from various sources as input data and index them as structural schema data. Then, they perform query-like programming that's similar to SQL on that data.

Inaccurate Capture and Replay

Here we discuss capturing logs on a production system and replaying or aligning them on a test system for testing and diagnostics. BDS components talk to each other, with their subcomponents distributed through a cluster of computers. Additionally, each component might have multiple processes and threads running in parallel, increasing the complexity. A busy BDS generates large volumes of logs with high velocity.

Capture

The larger the volume and velocity, the greater the observer effect, in which measuring a system's attributes changes the system. For example, when someone measures tire pressure using a tire pressure gauge, some air escapes from the tire, changing its pressure.

In BDSs, tracing mechanisms slow down the system because extra resources must be allocated to capture and store the log. The higher the workload intensity, the greater the observer effect because more resources are needed to capture the activities. This becomes especially important when you're trying to capture data for a heisenbug—for example, a timing-related one. When the system slows down, the timing problems might disappear because the chance of race conditions, deadlocks, and so on decreases.

So, it's important to build capturing infrastructure that minimizes the observer effect. Essentially, that infrastructure shouldn't significantly slow down the BDS.

Both software- and hardware-based solutions exist. Typically, software solutions are more prone to the observer effect but are more universal. Hardware solutions tend to be less intrusive but are platform-specific.

Software-based logging. These solutions fall into four categories.

First, many OSs incorporate log capturing. For example, the DTrace framework captures a program's execution path in real time. Developers don't need to modify the code to enable this instrumentation. It's part of the BSD kernel, making it available on Solaris, Mac OS X, FreeBSD, and NetBSD. An unofficial port also exists from DTrace to the Linux kernel.

Second, compiler-based tools capture code execution information. For example, Intel Compiler Code Coverage or GNU gcov can capture information about executed code blocks. However, this function requires recompiling the code. This tool's overhead is low, but the captured data is limited because information is lost about the sequence in which code blocks execute and about what data is passed from code block to code block.

At the other end of the spectrum, tools such as Intel Parallel Studio (IPS) capture information from multithreaded programs, track the state of shared memory, and so on. IPS is extremely useful for capturing and diagnosing problems in multiprocess and thread environments. Unfortunately, its observer effect is pronounced; performance degradation can reach multiple magnitudes.

The third category is custom-built solutions, which vary widely. Developers build logging infrastructure from scratch or reuse language-specific logging libraries (such as Apache log4j). So, they must manually instrument the code with probe points, specifying the information to capture at every point. Typically, probe points are near entries and exits to the functions and near important branching points.

Finally, specialized solutions can capture specialized types of logs (use-case-specific events) and are less universal than the other software-based solutions. For example, for databases, tools such as IBM InfoSphere Optim Workload Replay and Oracle Database Replay capture workloads on a production system and replay them on a test system to ensure accurate system testing. The tools work with minimal intrusion and slowdown and can often be configured so that the workload information is read directly off network cards. However, they won't capture low-level information about what's happening in the database engine.

Hardware-based logging. Capturing hardware-level log information minimizes the observer effect. Also, information is available at a very low level (often at the CPU instruction level). For example, IBM's z/OS can capture system- and transaction-level logs. Intel has been working on building processor-level tracing into its products,[11] but no commercial offering exists yet.

Replay

Regarding log replay, there are currently more questions than answers. As we discussed before, specialized tools for databases allow both workload capturing and replaying. However, they focus on relational databases.

Developers need tools that can capture intensive BDS production workloads and replay them on the test system in the presence of data obfuscation. In addition, they need tools for other BDS components such as the business intelligence layer. In many situations, you might need to generate logs that crosscut the layers of the entire software (or even hardware) stack. The need also exists for a general strategy for scaling down workloads. An alternative solution would be to dedicate extensive computing power to process large workloads. For example, if the test system is one-tenth the size of the production system, should the workload be reduced by nine-tenths (in terms of the number of concurrent connections, operations per unit of time, and so on)?

Once logs are captured and uploaded, they need cleaning to ensure their veracity.

Inadequate Tools

The volume problem manifests itself in not only the large volume of BDS-generated data but also the large volume of BDS component source code that requires instrumentation.

Enterprise-level software consists of millions of lines of source code; not every tool can handle such volumes. Typically, this will be manifested by a crash of the instrumented code, incorrect results, dramatic performance degradation, and so on. These symptoms can have various causes—for example, the observer effect, poor scalability (overflows in internal tool data structures), and incorrect code instrumentation (due to issues with a tool's code parsers).

Tool vendors are typically open to fixing the problems. Unfortunately, BDS component developers might be unable to share their source code owing to nondisclosure agreements. In this case, BDS developers must build self-contained test cases for the tool developers, which is laborious.

Performance degradation caused by log-capturing tools might exacerbate the observer effect. In addition, customers might not permit the use of such slow tools with their production systems.

Even with in-house test systems, performance might be important. Performance enables quick time to market. For example, we might run nightly regression tests in parallel on 100 computers to complete test executions by morning. Capturing the execution logs helps to diagnose automatically the test failures' root causes. However, even a 50 percent performance degradation (due to code instrumentation) will require 50 additional computers for regression-test runs, increasing test costs significantly.

Incorrect Log Classification

Once logs are captured and uploaded, they need cleaning to ensure their veracity. Also, only the required logs must be kept for analysis and classification because not all logs have value.

For example, imagine you've collected a log to diagnose a problem but are uncertain whether the log captured the problem's root cause. Multiple reasons exist for why the problem can't be captured reliably. In the simplest case, when the software crashes and a stack dump is gathered, you can usually capture the root cause on the first try (unless you have an extremely pathological case leading to stack corruption). However, if you're trying to capture an intermittent defect's root cause, you might need to run a test case multiple times. For example, the problem could be that the code contains a heisenbug and the “stars didn’t align” to trigger it the first time. Alternatively, you might need to run the test case multiple times because log-capturing tools for some BDS components must be disabled (to minimize performance degradation). By Murphy’s law, these are 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 because you often won’t know which logs contain defect patterns (and have value) and which don’t (and are worthless). Without this information, the classification models’ accuracy decreases significantly because all logs for a given problem must be classified as containing information about root causes (even though this isn’t always the case), confusing the classifier.

Such cases often require manual intervention. Developers must manually eliminate the worthless logs, often with the help of domain experts. Someone must be responsible for maintaining the data repository in a clean and consistent state. This person must follow up with developers and testers, ensuring that they keep only valuable logs.

A Variety of Log Formats

As we discussed earlier, BDSs consist of multiple software and hardware components. Even though universal log formats exist,[3,12,13] they aren’t widely used. So, these components will emit heterogeneous logs in a variety of formats.

For analysis, all logs must be converted to one unified representation. No universal converter exists, so log analysis tool developers typically build converters for each data format.

In addition, current universal formats have limitations. For example, the Knowledge Discovery Metamodel is an Object Management Group standard that supports tracing but doesn’t support any compaction mechanism.[12] In other words, a log will be represented in its original format, which hinders scalability.

The compact trace format (CTF) is a metamodel that models log information compactly (compacted logs can be compared without being uncompacted).[4] However, it’s limited to simple function call traces. CTF uses a graph theory concept to turn a routine call tree into an ordered directed acyclic graph (DAG) by representing similar subtrees only once. This way, a log should never be saved as a tree but rather as a DAG. CTF’s creators showed that this compaction scales well to large logs. But although researchers have extended CTF to support multiprocess systems,[4] it still isn’t expressive enough to be widely deployed. For example, it doesn’t support function arguments and statements.

Creating a universal log format to support all kinds of logs is challenging, and such a format still might be ineffective. This is because the need for a log type depends on the application domain. For example, system call logs are used extensively in security (particularly in anomaly detection) and in understanding how applications interact with the OS. These logs might 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. Another solution would be to develop analysis techniques that operate on common data such as event time stamps. However, we anticipate that such analyses are for limited purposes.

Inadequate Privacy of Sensitive Data

Owing to space constraints, we briefly highlight key issues related to the veracity of sensitive data captured in a log—for example, usernames, passwords, and credit card numbers.

The first issue concerns system debugging. You can use logs from a production system[14] to reproduce a bug exposed on the production system in a lab environment, as we discussed earlier. Existing bug reproduction techniques instrument the system to capture objects and other system components at runtime. When a faulty behavior 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 might contain sensitive information, causing customer privacy issues.

One solution to this issue is to use obfuscation and anonymization. Some logging frameworks have rule-based extensions that obfuscate or remove sensitive data before storing it in the log. However, such an approach isn’t error-proof because it depends on the accuracy of detection rules, which often change as the BDS source code evolves. Perhaps one day a tool will automatically and accurately recognize sensitive data and obfuscate it at runtime.

Another issue is that, owing to legal constraints, a user might be unable to share a log with a BDS manufacturer (as is the case for financial or military organizations). This issue has two solutions.

The first is to provide the client with a standalone version of a log

ABOUT THE AUTHORS

ANDRIY MIRANSKYY is an assistant professor in Ryerson University's Department of Computer Science. His research interests are in mitigating risk in software engineering, focusing on software quality assurance, program comprehension, software requirements, big-data systems, and green IT. Miranskyy received a PhD in applied mathematics from the University of Western Ontario. Contact him at avm@ryerson.ca.

ABDELWAHAB HAMOU-LHADJ is an associate professor in the Department of Electrical and Computer Engineering at Concordia University, Montreal. His research interests are software engineering, software tracing, mining execution traces, software maintenance and evolution, and trace-based anomaly detection systems. Hamou-Lhadj received a PhD in computer science from Ottawa University. Contact him at wahab.hamou-lhadj@concordia.ca.

ENZO CIALINI is a senior technical staff member for analytics in IBM's Worldwide Core Database Technical Sales organization and was the chief quality assurance architect for DB2 and PureData at the IBM Toronto Laboratory. His research interests are in software engineering (process, development, quality assurance, and so on). Cialini received a BSc in computer science from McMaster University. Contact him at ecialini@ca.ibm.com.

ALF LARSSON is a senior specialist in observability for Ericsson. His research interests are data mining, machine learning, big data, and diagnostics. Larsson received an MSc in computer science from Uppsala University. Contact him at alf.larsson@ericsson.com.

processing and analysis tool. This tool should be able to analyze the log remotely, without any external information. This solution's feasibility depends on the storage and computing power the tool requires.

The second solution is to use homomorphic encryption on the log so that the BDS manufacturer can analyze it without being able to extract sensitive information. However, this solution is computationally expensive and isn't practical yet.

The issues and solutions we discussed here should be of interest to practitioners because they can readily leverage existing techniques to build their own solutions. Our findings should also be of interest to the academic community because they highlight unsolved practical problems.

References

1. A. Mockus, "Engineering Big Data Solutions," *Proc. Future of Software Eng.* (FOSE 14), 2014, pp. 85–99.
2. T. Reidemeister et al., "Diagnosis of Recurrent Faults Using Log Files," *Proc. 2009 Conf. Center for Advanced Studies on Collaborative Research* (CASCON 09), 2009, pp. 12–23.
3. R. Brown et al., "STEP: A Framework for the Efficient Encoding of General Trace Data," *Proc. ACM SIGPLAN-SIGSOFT Workshop Program Analysis for Software Tools and Eng.*, 2002, pp. 27–34.
4. A. Hamou-Lhadj and T.C. Lethbridge, "A Metamodel for the Compact but Lossless Exchange of Execution Traces," *Software & Systems Modelling*, vol. 11, no. 1, 2012, pp. 77–98.
5. H. Pirzadeh et al., "Stratified Sampling of Execution Traces: Execution Phases Serving as Strata," *Science of Computer Programming*, vol. 78, no. 8, 2013, pp. 1099–1118.
6. A. Oliner, A. Ganapathi, and W. Xu, "Advances and Challenges in Log Analysis," *Comm. ACM*, vol. 55, no. 2, 2012, pp. 55–61.
7. S.S. Murtaza et al., "An Empirical Study on the Use of Mutant Traces for Diagnosis of Faults in Deployed Systems," *J. Systems and Software*, Apr. 2014, pp. 29–44.
8. L. Mariani, F. Pastore, and M. Pezze, "Dynamic Analysis for Diagnosing Integration Faults," *IEEE Trans. Software Eng.*, vol. 37, no. 4, 2011, pp. 486–508.
9. A. Kuhn and O. Greevy, "Exploiting the Analogy between Traces and

Signal Processing," *Proc. 2006 IEEE Int'l Conf. Software Maintenance*, 2006, pp. 320–329.
10. A.V. Miranskyy et al., "SIFT: A Scalable Iterative-Unfolding Technique for Filtering Execution Traces," *Proc. 2008 Conf. Center for Advanced Studies on Collaborative Research* (CASCON 08), 2008, article 21.
11. J. Reinders, "Processor Tracing," *Intel Developer Zone*, 18 Sept. 2013; https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing.
12. *Information Technology—Object Management Group Architecture-Driven Modernization (ADM)—Knowledge Discovery Meta-Model (KDM)*, ISO/IEC 19506:2012, International Org. for Standardization, Apr. 2012; www.iso.org/iso/catalogue_detail.htm?csnumber=32625.
13. G. Lee et al., "The Unified Logging Infrastructure for Data Analytics at Twitter," *Proc. VLDB Endowment*, vol. 5, no. 12, 2012, pp. 1771–1780.
14. H. Jaygarl et al., "OCAT: Object Capture-Based Automated Testing," *Proc. 2010 Int'l Symp. Software Testing and Analysis*, 2010, pp. 159–170.