Provenance from Log Files: a BigData Problem

Devarshi Ghoshal and Beth Plale
Indiana University
Bloomington, IN 47405
{dghoshal,plale}@cs.indiana.edu

ABSTRACT
As new data products of research increasingly become the product or output of complex processes, the lineage of the resulting products takes on greater importance as a description of the processes that contributed to the result. Without adequate description of data products, their reuse is lessened. The act of instrumenting an application for provenance capture is burdensome, however. This paper explores the option of deriving provenance from existing log files, an approach that reduces the instrumentation task substantially but raises questions about sifting through huge amounts of information for what may or may not be complete provenance. In this paper we study the tradeoff of ease of capture and provenance completeness, and show that under some circumstances capture through logs can result in high quality provenance.

1. INTRODUCTION
Provenance by definition means origin or source. Provenance is familiar in its use to describe the lineage or ownership of a work of art. When applied to data, provenance is the lineage of a data set, capturing transformations or derivations applied [3,22,33]. As research increasingly depends on contributory data from sources outside the research team, the need for provenance of data grows correspondingly in order to facilitate trust in reuse of data. Similarly, as data grow larger, the analysis carried out to produce new knowledge requires more complex distributed and parallel processing. Capturing the nuances of execution environment that could influence the results of large scale data analysis is within the scope of provenance. Big data and increased data sharing, together impose a challenge to data provenance to become easier to deploy, easier to understand, and easier to make use of in determining the quality of the results of big data analysis.

But capturing provenance can be a burdensome and labor intensive task. Provenance capture is often integrated into a workflow system [19,27] which while an elegant solution, is only as useful as is the workflow system. One can add hooks to an application to capture provenance, called program instrumentation but this requires access to source code of every application for which provenance has to be captured. Moreover, source code instrumentation can lead to perturbation in execution which can affect outcomes if too high [31].

Logs are traditionally used for audit and for identifying root causes of failures in large systems. But, logs also contain important information about the events within a system that result in generating data objects. It has been shown that intelligent logging and careful analysis of logs can be used to extract key information about the system [28].

Since provenance is event based, capturing the important events within a system, event logs could be an important source of provenance information. Correlating and capturing provenance events from both structured and unstructured logs requires careful analysis of event logs. This paper explores the option of deriving provenance from existing log files, an approach that reduces the instrumentation task substantially but raises questions about the completeness of the provenance.

In this paper we study the tradeoff of ease of capture and provenance completeness. Specifically, we propose a model for identifying and collecting provenance data from logs and a framework for rule-based provenance identification and capture, from log files based on the model. Capturing provenance from log files has major advantages over instrumentation- a) it does not incur any execution perturbations due to source code instrumentation, and b) it does not require access to source code of the applications. Using a rule-based provenance collection mechanism also enables the user to refine rules for selective filtering of relevant information. It also provides seamless mechanism to integrate and correlate logs for distributed applications. We show that under some circumstances provenance capture through logs can result in high quality provenance.

The rule-based provenance collection framework we developed has two phases- a) event capture, and b) provenance derivation. Both the phases are controlled by specifying rules- for capturing raw provenance data from logs and transforming them into structured provenance events. Using XML based rules enable seamless mechanisms for rule serialization, interchange and embedding into web service models.

There are many aspects of using log data to collect prove-
nance information. Issues include understanding different levels of information that can be stored in log files, relevance of log data, and analyzing the trade-offs between source code instrumentation and log data analysis.

This paper makes the following contributions:

- Model for relations among log levels contributing to different granularity and type of provenance
- Framework to extract relevant provenance information from what might be massive logs
- Analysis of correlations between applications, log-levels, and the types of provenance information
- Assessment of provenance capture from log files, and the quality of provenance collected from log files

The remainder of the paper is organized as follows. In Section 2 we discuss related work. Section 3 gives a model for provenance capture from log files whereas in Section 4 we provide a general rule-based framework for provenance collection from log files. We show experimentation results in Section 5. We discuss our experimentation results and analysis of provenance capture from log files in Section 6. Finally, we present our conclusions in Section 7.

2. RELATED WORK

Provenance as a valuable addition to metadata has been studied for e-science workflows [11, 21], file systems [24], semantic web [6, 20], clouds [25] and databases [4, 9]. The use of provenance in determining the quality of scientific data and data provenance has also been shown [32, 7].

Solutions to provenance capture include providing services for provenance identification and storage [14, 5], provenance-aware solutions [36, 24], application or workflow instrumentation [19], and language extensions for provenance identification [8].

With respect to provenance harvesting from log files, SherLog [38] analyzes source code by leveraging large system logs. LogMaster [39] identifies event correlations to build failure correlation graphs by mining system logs. NetLogger [15] collects and analyzes event logs for performance of distributed applications but needs source code instrumentation. Also, it does not explicitly collect provenance information from log files.

Xu et al. [37] proposes a mechanism to mine logs combined with the source code that generated the logs to detect problems in large scale systems. Logsurfer [30] uses rules to monitor system logs. It has also been shown that careful logging and examination of system logs can result in generating important information [28]. Jiaang et al. [17] shows how combining failure messages with event messages can improve root cause analysis of failures in large systems. Process mining [23] techniques are applied to business processes for process discovery, conformance checking and enhancement from a system’s event logs using an XML-based standard for generating event logs (XES). But there is no framework for collecting provenance information from different types of logs from large scale distributed applications.

Whereas logs contain relevant information for error analysis, they also contain important information about the runtime execution of an application and generation of data objects at different steps in the program. While most of the log analysis in past has been done to understand root causes of failures, log analysis has not been done for extracting provenance information. Our technique is to leverage log data to extract and build provenance graphs so that the quality and process of data generation can be easily understood without any source code instrumentation.

3. MODEL OF CAPTURE

We introduce a model that guides the analysis of provenance capture from application logs. The model, shown in Figure 2, has three vertices: application type, log level, and provenance type. Logging information at the right log level is important for capturing relevant provenance. Through the model application logs with different levels of information are classified into different classes of provenance types.

We describe each vertex in the subsections that follow. The application use cases are used to develop outcome based assertions about log-level, provenance-class, and application-type.

3.1 Log Levels

Logging frameworks like log4j, log4net, nlog, python logger [2] support multiple logging levels. These levels are used to rank the importance of log messages and control the amount of information to be dumped into log files. Debugging levels are typically used during development of an application. An application once in production will use a non-debug logging level or a combination of these levels for logging runtime information about the system. Log level can be set at runtime through a configuration parameter. If a logger uses a lower level of logging (and hence, lower priority) then all the messages with same or higher levels are logged into the logs. The common logging levels in ascending order of priority are as follows:

- **TRACE** is the most verbose logging level. It is used for debugging and captures instruction level details such as a change to a particular register.
- **DEBUG** is used for debugging an application just as TRACE is, but at the high level programming language statement instead of machine instruction level.
- **INFO** is used for providing an overview of the debugging information which can be described in a single line rather than providing hex dumps or detailed information of statement execution.

Figure 2: Application types, log levels, and provenance type form the three dimensions of the provenance capture model.
most useful for monitoring and managing an application during execution. For instance, an INFO message will describe the various entry and exit points in the application. The WARN level is used for handled exceptions which can lead to potentially harmful situations. Examples include using of default configuration settings within a program. The ERROR level designates exceptions that result in failure of parts of the application but the application would still continue to run. These messages are often un-handled exceptions. The FATAL level is used for logging severe error events that might result in aborting the application. Both TRACE and DEBUG levels are used during program development and testing. The remaining four levels, INFO, WARN, ERROR, and FATAL, are used in a production setting.

- **Exception**: This level of logging is specific to failures and errors. It is used to log messages in WARN, ERROR, and FATAL.

Figure 3 shows a classification of log levels into provenance types. It also describes the relevant sections of log data necessary for contributing to a provenance class. The log level and provenance type shows the type of provenance data that can be obtained by using the messages in different levels of log files. Log data shows event information that has to be stored in and/or retrieved from the logs.

### 3.2 Classes of Provenance

Simmhan et al. [33] shows two major classes in provenance: a) **process provenance**, and b) **data provenance**. A provenance-aware system generally captures either or both types of provenance information. ‘Process provenance’ describes the process that generated data for which provenance is captured. It stores metadata about both the process and the data related to the process’s execution. On the other hand, ‘data provenance’ is treated as metadata associated with the derivation history of the data items involved in a process execution.

Furthermore, we can classify provenance data along the dimensions of prospective provenance and retrospective provenance [35, 12]. **Prospective provenance** is related to capturing a computational task’s ‘specification’ and corresponds to the steps that must be followed to generate data product(s). **Retrospective provenance**, on the other hand, captures the steps executed as well as information about the environment used to derive a specific data product.

### 3.3 Application Cases

We give a couple of application use cases that fall into distinct application classes. In each we collect provenance from the application log files. All the information captured is transparent to the application and collected through logs. This work evolved out of the Netkarma project [26] which
collects provenance information from applications running on a distributed testbed such as PlanetLab [10].

**Job Deployment to Distributed Infrastructure.** Job submission software deploys an application into a distributed or parallel environment. The job submission tool is a useful place to capture provenance because much is known about what components are run where. Sometimes the job submission tool carries out monitoring during execution too, to monitor progress. We experimented with provenance capture of GUSH [16], a shell-based job deployment tool used to deploy jobs to PlanetLab. GUSH accepts an XML document from the user describing resource needs, and other functions related to distributed application management. Commands can also be issued interactively at the shell to monitor certain characteristics of the distributed environment, including the hostnames, processes on hosts, uptimes etc.

Since, GUSH is a job submission tool, the log information for GUSH logs have two different views- one from GUSH’s perspective and another from the perspective of the job. From GUSH perspective, the logs record information at the fine-grained and exception-level. The log contains details about the steps performed by GUSH for deploying and executing the jobs. It also logs any information related to the failure during the deployment and execution of the job. From the job perspective, the log information is recorded at a more coarse-grained level. This is because the internal details of the job are unknown to GUSH. It also provides a causal ordering of events leading to the deployment and execution of experiments, and the lineage of data items used and generated by them. Due to the different perspectives, we capture two types of provenance information from GUSH logs: we capture both data and process provenance about the environment whereas we capture only data provenance about the jobs. We capture provenance information describing how the environment was setup for running a job, what job was executed, what are the components of the job, and what was the job execution status. The data provenance about the job includes information about what and how the data products were used and generated by it.

**Network Middleware.** Phoebus [29] is a network protocol for improved performance of data transfers over long distances. Phoebus builds off of GridFTP [1] which provides transfer monitoring. Provenance capture from Phoebus captures and records every transfer between hosts, transfer performance, effect of different protocols, characteristics of every transfer including the size and duration of transfer, and characteristics of the backbone network and hosts.

Phoebus logs do not contain internal details of the processes involved in transferring the data. They contain coarse-grained information. So we capture data provenance with respect to the transferred data between hosts.

The organization of log capture into relationships between log levels, provenance types, and application classes, provides a way of thinking about how to capture provenance from logs. But the model is not enough. The next piece is defining a set of rules that guide capture. We lay out a rule-based approach to create what we called ‘adaptors’ that parse raw data from logs into structured provenance events. Adaptors use a set of rules to correctly map raw provenance data into meaningful and structured provenance events.

### 4. RULE-BASED PROVENANCE CAPTURE

Provenance extraction from log files is carried out by code we call an “adaptor”. A set of rules are used to guide construction of an adaptor. In this section, we describe the rule engine, and the framework for processing raw log data into structured provenance events.

#### 4.1 Rule Engine

The rule engine uses a rule specification file written in XML. The rule specification uses a grammar for rules for both processing raw log data and generating structured provenance events. The grammar specifies three major types of rules: a) match-select rules, b) link rules, and c) remap rules. Figure 1 gives an example of a sample rule specification.

1. **Match-Select Rules**: are required for selecting a data value based on some successful matching criteria. These are straight-forward rules specifying how to select different objects of a provenance event based on any matching condition. They are also used to select any description of a provenance event.

2. **Link Rules**: specify links or relationships among the objects of different provenance events. These are required to identify how one object in a provenance event is associated with another provenance event. These are useful for linking two indirectly linked objects together (for example, a process being both an invoker and a producer).

3. **Remap Rules**: specify mechanisms for aliasing different objects in a provenance event. These are useful in cases when the objects in a provenance event are to be defined more meaningfully than how the event describes it in general. For example, a process-id can be remapped to the actual process name, or a host-ip can be remapped to a hostname.

The example rule specification in Figure 1 shows rules for generating a data-produced event-type. It also shows how one link object, PROCESS, for a provenance event provides an alias for PID, describing another object in a provenance event (remap rule). The selection of different levels of log messages is controlled by specifying appropriate values in the rule specification.
4.2 Framework

The architecture of the rule based provenance collection system shown in Figure 4 consists of a Rule Engine, Log Processor and an Adaptor.

**Rule Engine.** The rule language is implemented in XML. The rules for selecting information from the logs is specified through an XML document. The rule engine is integrated with the log processor and when executed, applies rules to the log data.

**Log Processor.** The log processor parses log files based on some specified rules and selects relevant information. It converts every matching log data into a set of vertices (or entities/objects) and edges (or relationships/dependencies). The log processor uses the rule engine to apply relevant rules to the log data to parse and select raw provenance data from the logs. It has two options for every log file processing. The first option treats every processing of a log file as a separate instance for provenance collection. The second option considers a single log file as a single source of provenance information.

**Adaptor.** The adaptor is responsible for mapping the raw provenance events into structured events. It builds subgraphs using the set of vertices and edges resulting into a graph of structured provenance events. It then encapsulates the information into serialized XML messages to be used by a provenance service. We used the Karma provenance collection tool [5, 34, 18] for storing, managing, and querying provenance information. The adaptor uses the open source RabbitMQ message bus for sending structured provenance events to Karma.

In summary, an XML-based rule language is used for selecting attributes from logs and for defining mapping rules between the unstructured raw log data and structured Karma-compatible provenance events. The Log processor parses log files and selects relevant data based on the rules. The Adaptor maps the results of the Log processor, into Karma provenance events, and sends it to the Karma server through the RabbitMQ Message bus. Karma provenance tool and then, processes each of those events and stores them in a relational provenance repository.

5. EXPERIMENTAL RESULTS

In this section, we carry out experimental analysis to understand the performance of the framework. We measure under log files of varying sizes and information complexity. We show that the amount of provenance information does not necessarily depend on the size of log files.

All the tests are performed on an Intel Xeon E7540 @ 2.00 GHz machine with 4 CPUs, and 6 cores/processor. It has 64-bit Red Hat Enterprise Linux Server release 5.8 and 128 GB memory. We use Karma version 3.2.0 with RabbitMQ message bus for provenance storage and retrieval. The tests are performed for logs varying in size from 100 KB to 100 MB.

We process log files of over 100 MB in less than a minute. These large log files are for a deployment and execution of a Twister map reduce application [13] running a breadth first search on graphs of 2000 vertices. The largest log was generated by running breadth first search for 10 graphs of 2000 vertices each (bfs-5000-10). The (bfs-2000-10) run also included a failed instance of the search algorithm. The smallest run of the breadth first search (bfs-2000-1) was only on one graph of 2000 vertices. One other application was listing the set of files on remote hosts (ls). Another application used was to read the data and count the number of lines of certain files on remote hosts (wc, cat). Finally, some logs were generated by issuing simple monitoring commands like ps, uptime etc. at the GUSH shell (ps). The log file set also contains logs generated by running these applications on different number of host machines. Figure 5 shows the execution time of processing these log files. The figure shows a low overhead for processing the raw data from log files and converting them into structured provenance events.

We collected 62,000 provenance events from an unstructured log file of 100 MB by using our rule based collection technique. We collected both data and process provenance from GUSH logs. For coarse-grained Phoebus logs, we captured data provenance for the data transferred between multiple hosts.

We collected 62,000 provenance events from an unstructured log file of 100 MB by using our rule based collection technique. Figure 6 shows that relatively large amount of provenance information can be collected from application logs without instrumenting the source code of the applications we considered. It should be noted that an increase in log size does not necessarily mean increase in provenance. The number of provenance events depends on the semantic richness of the logs and specifications of rules for identifying provenance.

For the results mentioned above, we used fine-grained and exception-level logs. We collected both data and process provenance from GUSH logs. For coarse-grained Phoebus logs, we captured data provenance for the data transferred between multiple hosts.

6. DISCUSSION

There are several challenges to using logs for capturing
provenance information. One of the major challenges is that it is impossible to capture provenance of any data for which no log information is available. Another challenge of using log files in a distributed environment is the mechanism of consolidating the logs from different modules and hosts. Also, an important consideration for collecting provenance from log files is to manage uniqueness and reduce redundancy in the data which we achieved by writing complex rules for selection and filtering. Finally, since provenance is a quality artifact it is important that the quality of the provenance data itself should be quantified.

In this section, we assess provenance capture from log files, assess level of effort against other provenance capture mechanisms, and assess the resulting quality of provenance. We also make recommendations for the researcher who wants to adopt provenance capture through log files into their applications as to what they can do to increase the odds of having higher quality provenance.

### 6.1 Provenance from Existing Logs

It is difficult to instrument legacy systems for provenance capture because source code is often inaccessible or too complex to instrument. In such cases turning to the log file could be a way to make provenance accessible. An important technique in such cases is to understand the way events and messages are classified among different levels in log files.

Collecting provenance from existing log files has its own limitations and challenges. Logs from large-scale applications often contain substantial information. A log file for our job deployment to distributed infrastructure use case is 1.6 million lines in length and generated approximately 100 MB of raw log data. The reason for such large log files is due to large amount of computation using and generating huge amount of data. The size of the log files also increase due to fine-grained logging. This information overload makes it difficult to identify and capture the relevant provenance information without a proper framework.

Where log file size is large, a rule-based system eases the task of provenance extraction because it allows for a flexible way of selecting relevant information and also gives control on managing the granularity of provenance information. The rule based approach provides mechanisms to select different levels of log data by using relevant match-select rules.

Another challenge for provenance extraction from logs is the correlation and linking of provenance information from multiple logs of a distributed application. One way to correlate provenance from different logs of a distributed application is to set a global context ID unique to each execution instance of the application. Another way is to batch log files, processing all log files from a single execution as a single batch. Whereas the former approach requires modifying the logging mechanism, the latter approach requires additional scheduling mechanisms to batch-process the logs.

Controlling redundancy by duplicate processing of log files is a major challenge. We treated every log processing as a unique instance and assigned a unique ID for every processing. For removing duplicates from a single file we used rules to selectively filter out duplicates.

For logs that do not contain any provenance information, the best approach is to modify the logging mechanism or code instrumentation.

### 6.2 Provenance from Logs of New Applications

There are best practices for writing to logs. For new applications, it is important to structure the logs in such a way that provenance information can be captured at different granularities. Figure 7 shows how we classified the provenance information from different applications by using different levels of log data.

Since fine-grained and exception-level logging contain more detailed information about process details, they should be used to log process provenance information. Data provenance information can be captured through both fine-grained and coarse-grained logging.

Provenance is also useful in detecting anomalies and errors in program execution. Log level exception should be used for capturing provenance information in such cases. Since these anomalies can be the result of both data and process fallacies, it is best to capture provenance at both data and process level.

### 6.3 Provenance Quality

Cheah et al. [7] propose an approach to evaluating the

---

**Table 1: Quality Measures for Provenance from GUSH Logs**

<table>
<thead>
<tr>
<th>Rule File Type</th>
<th>Objects/Applications</th>
<th>Generic</th>
<th>Enhanced</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bfs-2000-10</td>
<td>bfs-2000-1</td>
<td>ls</td>
</tr>
<tr>
<td>Entities</td>
<td>13</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Artifacts</td>
<td>19571</td>
<td>1341</td>
<td>3</td>
</tr>
<tr>
<td>Annotations</td>
<td>478</td>
<td>104</td>
<td>104</td>
</tr>
<tr>
<td>Causal Dependencies</td>
<td>61589</td>
<td>4398</td>
<td>14</td>
</tr>
<tr>
<td>Log Size (MB)</td>
<td>112.5</td>
<td>9.0</td>
<td>0.15</td>
</tr>
</tbody>
</table>

---

![Figure 7](image-url)
quality of provenance data by means of structural and contextual analysis of provenance data. Here we examine quality in the context of our application cases.

Completeness. The log semantics are different in GUSH for applications deployed and executed through a job description XML file compared to issuing commands at the shell interactively. We use a generic rule file for both kinds of logs. We then refine the generic rule file to create two enhanced rule files for each type. We compare the provenance data using both. Using the methods described in [7] we analyzed for entities, annotations, artifacts, and the causal dependencies. We observed that annotations were the most affected when the generic rule file was used. Table 1 shows the number of objects generated by using the two rule files. The (ps) column using enhanced rule file are proportionate when to the corresponding log file sizes. With an increase of approx. 10× in log size, the semantically same application generates approx. 10× more provenance data. But, this is not true for the generic rule file. The increase in provenance data is inconsistent using the rule file. It can also be observed that number of annotations are increased when the enhanced rule file is used. Further analysis shows that entity annotations are the ones which were missed using the generic rules. These attributes signify that by analyzing the graphs structurally, one can deduce that using the enhanced rule file, more complete provenance was captured as compared to using the generic rules.

Redundancy. Cheah et al. [7] suggests that duplicate provenance lowers the overall quality of the provenance. This is primarily because it can contribute to substantially larger volumes of provenance. The rule based provenance identification and collection mechanism we propose filters duplicate information prior to its being turned into provenance events, thereby eliminating duplicates in the resulting provenance.

Timestamp. Timestamps are optional in OPM and edges in an OPM graph can be extended with the optional timestamps, if available. Cheah et al. [7] showed that in a distributed environment, quality of provenance data can be affected if the causal dependencies do not comply with the time for cause and effect. In other words, if P1 happens before P2 then time(P1) < time(P2), which is not always true in a distributed setup. But for GUSH logs, there is a single observer (GUSH) which observes the events for both job deployment and execution in a distributed environment. So, we patched the original GUSH source to include timestamp information in order to validate the causality of events. We observed zero timestamp discrepancies for causal dependencies in GUSH logs. The timestamp information also helped in understanding the timed occurrences of events at different sites with respect to the observer. The physical timestamp information along with the logical timestamps provided a stronger basis of both time and causality in the provenance data captured through the logs. The timing information is also used to represent more meaningful visualizations.

Error and Failure. The (ps) column using enhanced rule file in Table 1 shows lesser entities than what was captured using the generic rule file. Further investigation showed that the generic rule file incorrectly identified an object as an entity which resulted in a ‘dangling’ entity. The same number of resulting artifacts and causal dependencies in both generic and enhanced rule files also suggest the same. As mentioned earlier, (bfs-2000-10) also has a failure instance of the search algorithm. Careful analysis of provenance information shows missing output data for one instance which implied failure in execution.

7. CONCLUSION AND FUTURE WORK

Automating provenance collection in large scale applications with minimal changes to the application or environment is a hard problem. In this paper, we proposed piggybacking onto log files created for other purposes to collect provenance and developed a framework for rule-based provenance collection from log files. We found in working with the GUSH job submission tool that the developers were amenable to receiving a code patch from us that added additional information to the logging; this patch specifically enhanced the quality of the provenance.

As to ongoing work, we are looking into using advanced rule engines like Jess for specifying rules for collecting provenance. A naive approach in contrast to a rule-based implementation, is to develop log parsers for every application log. A better approach is to generate logs in a single, unified structure with provenance events embedded in them. We will explore machine learning techniques as a way of identifying relevant provenance in a massive log file. We would also like to use our methodology for collecting provenance from NetLogger logs.

8. ACKNOWLEDGMENTS

This work was funded in part by the National Science Foundation through GENI Spiral 2 funding. We thank You-Wei Cheah and Peng Chen of Indiana University for numerous discussions. We also thank Chris Small from the Global Research Network Operations Center at IU.

9. REFERENCES


