Improving Reproducibility of Data Science Pipelines through Transparent Provenance Capture

Lukas Rupprecht
IBM Research–Almaden
Lukas.Rupprecht@ibm.com

James C. Davis
Virginia Tech
davisjam@vt.edu

Constantine Arnold
IBM Research–Almaden
Constantine.Arnold@ibm.com

Yaniv Gur
IBM Research–Almaden
guryaniv@us.ibm.com

Deepavali Bhagwat
IBM Research–Almaden
deepavali@us.ibm.com

ABSTRACT

Data science has become prevalent in a large variety of domains. Inherent in its practice is an exploratory, probing, and fact finding journey, which consists of the assembly, adaptation, and execution of complex *data science pipelines*. The trustworthiness of the results of such pipelines rests entirely on their ability to be reproduced with fidelity, which is difficult if pipelines are not documented or recorded minutely and consistently. This difficulty has led to a *reproducibility crisis* and presents a major obstacle to the safe adoption of the pipeline results in production environments. The crisis can be resolved if the *provenance* for each data science pipeline is captured transparently as pipelines are executed. However, due to the complexity of modern data science pipelines, transparently capturing sufficient provenance to allow for reproducibility is challenging. As a result, most existing systems require users to augment their code or use specific tools to capture provenance, which hinders productivity and results in a lack of adoption.

In this paper, we present URSPRUNG,[1] a transparent provenance collection system designed for data science environments.[2] The URSPRUNG philosophy is to capture provenance and build lineage by integrating with the execution environment to automatically track static and runtime configuration parameters of data science pipelines. Rather than requiring data scientists to make changes to their code, URSPRUNG records basic provenance information from system-level sources and combines it with provenance from application-level sources (e.g., log files, `stdout`), which can be accessed and recorded through a domain-specific language. In our evaluation, we show that URSPRUNG is able to capture sufficient provenance for a variety of use cases and only adds an overhead of up to 4%.

PVLDB Reference Format:
Lukas Rupprecht, James C. Davis, Constantine Arnold, Yaniv Gur, Deepavali Bhagwat. Improving Reproducibility of Data Science Pipelines through Transparent Provenance Capture. *PVLDB*, 13(12): 3354-3368, 2020.
DOI: https://doi.org/10.14778/3415478.3415556

[1]"Ursprung" means "origin" in German.
[2]We presented a demo of URSPRUNG at SIGMOD'19.

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415556

1. INTRODUCTION

A large fraction of industry and research nowadays has deployed data science pipelines to extract insights and assets from the vast amounts of available data. However, the exploratory and iterative nature of data science work makes it difficult to reproduce the results of such pipelines. This leads to a reproducibility crisis in modern data science, in particular in artificial intelligence and machine learning [40, 17, 43], as without knowledge of *how* a result was produced or model was trained, they cannot be trusted. One cannot repudiate [10], reproduce, reuse, or improve on previous results if their lineage is not understood. This lack of trust hinders widespread adoption, collaboration, and continued refinement.

Incorporating *provenance* into execution environments for data science pipelines offers a solution to address the reproducibility problem. A *provenance model* records and stores the *lineage* of the pipeline output, such as an analysis result or a machine learning model. This lineage tracks the origin of the output and the actions and actors that transformed it. For example, the lineage of a machine learning model may consist of the input data, the algorithms, the configuration parameters, the associated dependencies, the various transformations applied to the input, the number of iterations, the execution environment etc. [26, 38]. Additionally, it also includes actors such as the owner of the input data, the owner of the executables, or the users who initiated and reviewed the pipeline.

Combining the provenance of all data in a system forms the *provenance graph*. The provenance graph is well suited as the basis for investigating the provenance of different assets, e.g., a machine learning model. Within its vertices and edges, it captures the actors, the actions and the resulting transformations. A well annotated provenance graph can be utilized to understand, compare and contrast, and consequently reproduce a data science pipeline [32].

Though promising and sound in principle, pervasive adoption of provenance has proven challenging in practice. Capturing the required information is not straightforward, and the main challenge is the inherent trade-off between generality and level of detail, i.e., maximizing the signal to noise ratio, that provenance faces.

With current approaches, users can choose between two main options: 1) *generic provenance capture systems*, which interpose at the operating system layer to capture provenance at the granularity of system calls and IPC [56, 33, 60]. This precision provenance is difficult to make actionable without a deep knowledge of the internals of one's pipeline and its execution environment; 2) *model management systems* [54, 69, 71, 67, 8, 5, 9], which provide richer information but force users to modify their software or adopt a particular data science framework and tool suite. While pipeline-specific information is valuable, the vast and quickly-changing data science landscape suggests that shoehorning users into a particular system or methodology is impractical.

We address these problems in our solution: URSPRUNG, a practical implementation of a provenance collection, storage, and mining system for reproducing data science pipelines.URSPRUNG aims to maximize both generality and the level of detail while remaining transparent to users. Therefore, it taps into and combines provenance data from a variety of readily-available provenance sources. The key idea of URSPRUNG is to only support sources that are *already natively part* of the data science pipeline (e.g., log files or `stdout`) and the underlying compute and storage system (e.g., OS and file system). As a result, applications and pipelines can remain unchanged while enough provenance is captured to analyze, compare, and reproduce results.

URSPRUNG considers system-level sources such as system call notifications, and application-specific sources such as log files or databases. System-level provenance is collected through standard auditing mechanisms such as `auditd` [6]. URSPRUNG introduces a *rule language* to define capture rules for different application-specific sources, which offers an easy way to harvest relevant provenance information from these sources without application changes.

To increase practicality, URSPRUNG: 1) uses *event aggregation models* to merge fine-grained low level provenance into more compact, intuitive information to reduce resource overheads and improve queryability; 2) provides configuration through *provenance classes*, i.e., high-level abstractions to tailor provenance collection to a pipeline's needs; and 3) has an integrated interactive GUI with a differential provenance view [27] to explore, contrast and compare pipeline executions.

After introducing the relevant background on provenance and and its requirements on data science pipelines (§2), we make the following contributions:

- We introduce URSPRUNG, a provenance implementation tailored to enhance the reproducibility of complex data science pipelines (§3.1).
- We address challenges in provenance expressiveness and overhead in the form of a rule language to collect application-specific provenance from existing sources (§3.2), and provenance classes and event aggregation models to increase the practicality of the collected provenance data (§3.3).
- We describe the URSPRUNG GUI, an interactive GUI that supports a variety of features such as step-by-step graph exploration, file content comparison, and differential provenance, to ease the exploration of the provenance graph (§4).

We implement URSPRUNG and discuss practical implementation considerations (§5) and then evaluate it using four representative data science pipelines from real-world use cases and benchmarks (§6). The evaluation demonstrates that URSPRUNG is able to capture enough provenance to compare and reproduce different pipeline runs while only adding an overhead of up to 4% for pipelines consisting of a large number of small steps. For pipelines with fewer, larger steps, URSPRUNG's overhead stays within 1%.

2. PROVENANCE AND DATA SCIENCE

In this section we provide background on modern data science pipelines (§2.1) and study the requirements of provenance collection for such pipelines (§2.2). We then discuss the shortcomings of existing provenance solutions when applied in this context (§2.3).

2.1 Data Science Pipelines

The data science reproducibility crisis is fueled by the complexity of the underlying pipelines. Data science pipelines involve many steps covering disparate activities such as data integration, preparation, cleaning, and model selection [44]. Each step has parameters that may be explored and tuned to achieve the desired result. This leads to many iterations over the same data set and the use of different tools for different steps of the pipeline [53, 69]. Tracing these steps is important for several reasons. For example, quickly identifying errors and rolling back pipelines to a working state is critical in a business context [23] while in a scientific context, comparing the effect of different parameters is helpful for experimentation and ensuring the validity of the work. The need for in-depth, end-to-end tracking of complex data science pipelines has also been called out most recently as one of the main functionalities *Enterprise Grade Machine Learning* must offer [13].

However, tracing the detailed history for different pipeline executions is challenging as data science is often an unstructured, ad-hoc process [71, 67]. The problem worsens as the number of generated data sets [36] and the complexity of the pipelines increases [65]. For example, Microsoft's teams describe lengthy data science pipelines, starting with data cleaning and concluding with deployment and monitoring [15]. Companies like Google [23], Facebook [37], Pinterest [75], Booking.com [19], and AirBnB [35] maintain similar pipelines. These pipelines involve many tools and can involve a mix of automated and manual effort, which make them error-prone for both technical and sociological reasons.

2.2 Data Science Provenance Requirements

From our study of data science practices, both those described in the scientific literature (see §2.1) and those followed in-house at IBM Research, we believe that a provenance system for modern data science must meet the following requirements:

1. **Tailored Provenance**: Every data science pipeline is different, using different (and changing!) tools and following different processes. The particular provenance necessary to describe and reproduce the output of a pipeline will vary. A practical provenance system must allow users to tailor the provenance captured to suit their pipelines and their needs.
2. **Scalable Provenance**: Analytics clusters can often be large in size, spanning tens and hundreds of nodes. Provenance collection needs to be able to work in a distributed environment and scale to those cluster sizes.
3. **Lightweight Provenance**: Analytics pipelines may run 24/7 on terabytes of data. A provenance system must capture enough information to be useful, but need not be encyclopedic and should avoid storing an excessive amount of data.
4. **Usable Provenance**: Provenance systems are only useful if they can be used. They must not impose significant configuration requirements, nor are our data scientist colleagues willing to significantly modify their software in the pursuit of provenance. Once collected, the provenance must be presented in terms that a data scientist would understand.

2.3 Families of Provenance Systems

Provenance systems have been proposed for a range of computing contexts. URSPRUNG sits at the intersection of two families of provenance systems: *fixed-toolset capture systems* and *whole-system capture systems*. We introduce those families here and discuss why we believe they do not meet all of the above requirements.

The Fixed-toolset Capture Approach. Provenance systems in the fixed-toolset family support provenance capture for data science pipelines built using the toolsets that they support. *Workflow management systems* and *model management systems* both fall into this category. Scientific workflow management systems allow users to construct and execute processing pipelines through a specific interface (often graphical) [25, 48, 74]. During execution, these systems can capture provenance for a pipeline. Other researchers have considered workflow provenance for single-language pipelines, e.g., the Python-only noWorkflow provenance capture system [57, 63].

Rather than following the process-oriented approach and capturing provenance for workflows, model management systems focus on capturing provenance for some of their products — machine learning models. The idea was introduced by Kumar et. al. [44] and has been implemented in a variety of research projects [71, 53, 67, 69] and community projects [8, 5, 9].

Overall, these fixed-toolset systems are effective for capturing provenance for pipelines that use the tools they support. However, imposing a framework and toolset onto their users violates the Tailored Provenance and Usable Provenance requirements.

The Whole-system Capture Approach. We call the second family of interest *whole-system capture systems*. These systems aim to construct a provenance graph for every operation performed on a computing system, often for the purpose of security auditing. Some interpose at the level of the OS kernel [56, 33, 18, 60], while others consider subsystems like the file system [70, 66]. Because system users are typically uninterested in fine-grained information, the whole-system capture approach requires solving the *layer problem* [55]. These systems must handle system interactions at different levels of abstraction, e.g., interpreting the lower-level events they collect in terms of the higher-level semantics relevant to a user. The common solution is to require (possibly-automated [33]) changes to applications to add annotations or use a provenance library [55, 50]. This violates our Usable Provenance principle.

Due to the above reasons, we believe that the reproducibility use case for data science requires a different approach to provenance collection, which can meet all four major requirements (see §2.2).

3. THE URSPRUNG SYSTEM

We now describe the URSPRUNG system, which blends techniques from fixed-toolset and whole-system provenance capture systems. We first give an overview of the architecture (§3.1) and explain how URSPRUNG fulfills the four provenance requirements. We then discuss in detail how URSPRUNG enables *Tailored Provenance* through its rule language (§3.2), and *Lightweight Provenance* through provenance classes and event aggregation (§3.3).

3.1 Architecture

As shown in Figure 1, URSPRUNG consists of three main components: (i) the provenance sources; (ii) the collection system; and (iii) the provenance store and the GUI. The collection system itself comprises a base system and *provenance daemons* (`provd`) on each cluster node, which are responsible for configuring individual hosts for provenance collection.

Provenance Sources. The provenance sources provide the foundation of URSPRUNG and emit provenance-related *events*, which are captured and processed. URSPRUNG distinguishes between two types of sources: system-level and application-specific sources.

System-level sources are generic, application-independent components such as the operating or file system, and they generate low-level events such as system calls or file system interactions. This information is used to assemble the *base provenance*, i.e., basic information on creation and deletion of processes and data (files, objects, etc.) and their interactions.

There are different ways to extract relevant events from system-level sources. One option is to directly collect events in the OS kernel by augmenting the kernel with collection-specific code [56].

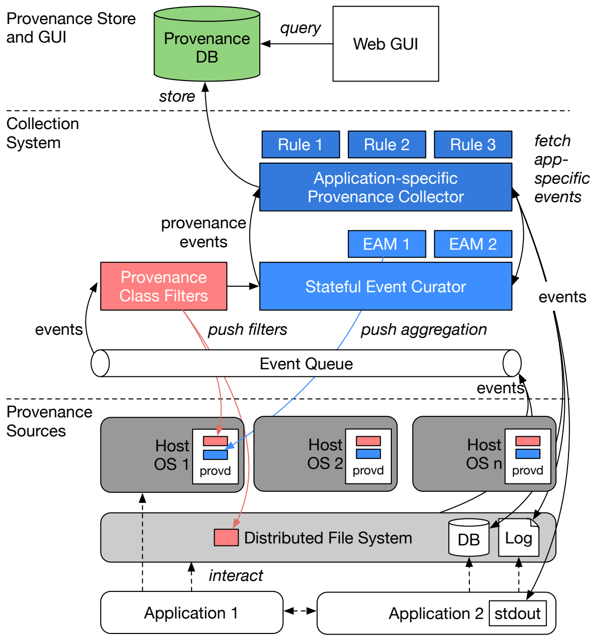


Figure 1: URSPRUNG architecture overview

Another way is to use existing hooks inside the kernel and once triggered, emit the corresponding event [60]. Finally, user-space applications can subscribe to a variety of notification mechanisms to receive provenance-relevant events, e.g., Linux's `auditd` [6] and `inotify` [7], or IBM® Spectrum® Scale's Watch Folders [3]. Learning from SPADE [33], we designed URSPRUNG to run in user space to avoid tight kernel dependencies. In particular, URSPRUNG uses `auditd` to receive system call notifications, and Spectrum Scale's Watch Folders for cluster-wide file system interactions.

Application-specific sources can be anything that an application depends on or produces during its execution. This includes sources such as log, config, and tmp files, any databases on which an application relies, and the application's command line output. This high-level provenance augments the base provenance information, providing more detailed insight into the operation of a pipeline [55].

URSPRUNG comes with a rule language to permit users to tap into application-specific sources (§3.2). These rules are triggered by base provenance events. For example, when URSPRUNG is configured to extract provenance from a temporary file f_{tmp}, URSPRUNG will monitor base provenance to detect write interactions with f_{tmp}. When writes occur, URSPRUNG will trigger the rule to extract provenance data from the new content. This addresses the **Tailored Provenance** requirement.

Collection System. The collection system is at the core of URSPRUNG and is responsible for consuming the events emitted by the provenance sources. It addresses the requirements of Scalable and Lightweight Provenance.

To provide **Scalable Provenance**, URSPRUNG employs an event-driven architecture. Each provenance source pushes its events to a scalable, distributed message queue, from which *consumers* pick them up. Our implementation uses the Kafka message queue [4], which some sources already support as a sink for events [3] while others can be configured to redirect events to Kafka through plug-ins [6]. Due to the message queue's distributed nature, URSPRUNG can collect events across all nodes in the cluster in a scalable way.

Additionally, the push-based approach helps URSPRUNG access *transient provenance* data with low overhead. Transient prove-

Table 1: Summary of URSPRUNG rules

Rule	Description	Syntax
FileLoad	Bulk load temporary data from file	`FILELOAD path INTO user:password@host:port/table USING col1,col2,...,coln`
LogTransfer	Extract relevant records from log files	`LOGTRANSFER path MATCH 'regex' FIELDS f1,f2,...,fn DELIM 'delimiter' INTO user:password@host:port/table USING col1,col2,...,coln`
DBTransfer	Transfer newly added data from application DB	`DBTRANSFER 'query'/'queryStateAttribute' FROMDSN 'dsn' TO user:password@host:port/table USING col1,col2,...,coln`
CaptureCout	Extract relevant records from a process' `stdout`	`CAPTURESOUT MATCH 'regex' FIELDS f1,f2,...,fn DELIM 'delimiter' INTO user:password@host:port/table USING col1,col2,...,coln`
Track	Track the content of matching files	`TRACK 'regex'`

nance, e.g., the `stdout` of a process or a temporary file, exists only during the execution of a pipeline and can be accessed through URSPRUNG rules. This means that, as the provenance is ephemeral and may only exist for a short period of time, these rules must be triggered quickly enough to extract all relevant provenance before it disappears. The push model ensures that events reach the collection system as fast as possible to allow timely event delivery.

To support **Lightweight Provenance**, URSPRUNG offers provenance classes and event aggregation models (§3.3). Provenance classes permit configuring which events are of interest and which events can be filtered, reducing processing and storage costs. They offer a simple abstraction for users to tailor collection to their specific needs. Event aggregation models define how different events can be combined to form denser, semantically richer events. Where possible, filters and aggregations are distributed to the provenance daemons to reduce load on the collection system.

Figure 1 shows how a raw event is processed by the collection system. If not filtered by the configured provenance classes, an event is passed to the *Stateful Event Curator* (SEC). The SEC has different *Event Aggregation Models* (EAM), which define, how to transform events into *provenance events*. While some transformations are local to a single event and hence, stateless, others are stateful as they require to combine several subsequent events. The ready provenance events are passed to the *Application-specific Provenance Collector* (APC), which matches the event against the configured rules. If there's a match, the APC will execute the rule and fetch any new application-specific provenance records, which are fed back to the SEC for potential aggregation.

Provenance Store and GUI. For consumption by the user, provenance events are persisted in the provenance database. Previous systems have utilized a variety of data stores for provenance data, including relational, XML, RDF, and graph stores [39]. URSPRUNG uses a relational database as its provenance store due to their scalability and fast query performance. The schema is optimized to allow for fast exploration of the provenance graph as part of the GUI by minimizing joins and not using expensive recursive queries. Hence, it is currently not compliant with PROV-DM [72]. However, PROV-DM or other schemas and stores can be supported by URSPRUNG by adapting the SEC.

URSPRUNG comes with a Web-based GUI that can be accessed from any browser. The GUI allows to interactively explore the provenance of files in a graph viewer panel and provides additional features to compare and analyze data science pipeline executions. To reduce complexity, it shields the actual database queries from the user and only allows a predefined set of queries through GUI interactions. It also permits configuring URSPRUNG through rules and provenance classes. In combination, the GUI satisfies the requirement of **Usable Provenance**.

3.2 Application-specific Capture Rules

To make provenance collection complete, it is necessary to augment system-level provenance with higher-level, application-specific provenance [55, 50]. To remain transparent and support a wider range of applications out of the box, URSPRUNG only relies on provenance that is already exposed through existing application-specific sources. We identify 3 main such sources: (i) files produced or used by the application, e.g., log, config, or temporary files; (ii) application-internal databases, which keep application state across runs of the application; and (iii) the command line output (`stdout` and `stderr`) of an application. File sources are further subdivided into files for bulk loading and files from which only specific parts should be extracted.

URSPRUNG's rule language provides generic capture rules for each of these provenance sources (see Table 1 for a summary). Each rule has conditions that govern when it should fire. Next we describe these conditions, and then the rules themselves.

Conditions. Conditions are used to prescribe the events to which a rule applies, and the circumstances under which it should be applied. They are Boolean expressions over the fields of an incoming provenance event, with support for arithmetic and regex matching.

FileLoad. The FileLoad rule is used to bulk load an entire file into the provenance database. This is useful when an application generates temporary data, e.g., in the form of a `.csv` file, which needs to be captured and stored for later analysis. The rule requires the connection string to the provenance store, and the table name and schema into which the data should be loaded. The `path` argument is a constant and describes that the path field of the event should be used to determine the file to be loaded.

LogTransfer. This rule enables watching an application file for new provenance-relevant records, such as log entries noting pipeline parameters. The `path` and database connection string are the same as in the FileLoad rule. The LogTransfer rule additionally requires a regex to indicate the entries of interest and the fields to extract from the entry. For files that follow append-only semantics, URSPRUNG will track the file offset so that only new content is parsed.

DBTransfer. The DBTransfer rule has the same purpose as the LogTransfer rule, except that the source is a database. URSPRUNG connects to this database via ODBC and requires the corresponding DSN. Additionally, users need to specify the `SELECT` query, which is used to extract new records from the source database. Again, URSPRUNG assumes an append-only table to avoid re-scanning the

```
path=/gpfs/spark/logs/*Master* AND written>0 AND
     event=CLOSE
->
LOGTRANSFER path
MATCH 'Registered app'
FIELDS 1,8,11 DELIM ' '
INTO dbUser:dbPassword@dbHost:dbPort/workflows
USING starttime,name,id
```

Listing 1: A LogTransfer rule to extract job information from the Spark master log

database and redundantly loading records each time the rule fires. Therefore, users need to provide the `queryStateAttribute`, a monotonically increasing attribute such as a timestamp or a surrogate key that is used as the offset into the (sorted) table.

CaptureCout. CaptureCout rules are used to extract provenance from the command line output of an application. Similar to LogTransfer rules, a regex and a format (fields and delimiter) need to be specified to identify the relevant provenance records. Once URSPRUNG detects that a process that should be watched has been started (by looking for `exec` events), it will transparently redirect the `stdout` and `stderr` streams of the process to a temporary file through `ptrace` and then watch the file for provenance data.

Track. Track rules allow URSPRUNG to support tracking file contents. This is useful to keep track of specific pipeline parameter or algorithm changes in source or config files. Each time a file is updated, URSPRUNG will copy the affected file to a private version control repository (Mercurial) and commit the new version. It also creates an entry in the provenance store to link the version hash to the update event. This rule uses a regex to identify the files to track. For best performance, this rule should be used preferably on small files to avoid high IO and storage overhead.

Listing 1 shows a sample LogTransfer rule to collect information from Spark jobs. The conditions direct URSPRUNG to watch the Spark master log and trigger the rule whenever the log file is closed after a write. The rule searches the log for the string 'Registered app' and extracts the fields 1, 8, and 11, corresponding to a job's start time, name, and ID, from matching entries. The triple is then loaded into the `workflows` table in the provenance database.

Rules offer data scientists a convenient way to tap into external, application-specific provenance sources. Compared to requiring application-level changes such as adding or annotating code to emit provenance from existing applications, URSPRUNG's rules are less intrusive and take less effort to set up. This is particularly useful in cases in which provenance needs to be extracted from large existing applications not written by the data scientist: understanding another codebase and instrumenting provenance-generating locations is both complex and time consuming. While basic understanding of the application behavior is required to write useful rules, such knowledge can often be retrieved from the documentation or by reading through examples. URSPRUNG can also offer pre-defined rules for the most commonly used data science applications.

3.3 Reducing Provenance Data

Provenance data can grow quickly, putting pressure on the provenance store and making queries expensive. This problem is inherent to provenance, but it can be staved off by reducing the data collected. Filtering and aggregation are two basic methods for reducing data. However, one must know what to filter and how to aggregate. This is often not straightforward, especially if crossing semantic gaps from system-level provenance to application-level provenance. To ease this task and effectively reduce the amount of collected provenance data, URSPRUNG uses the concepts of *provenance classes* for filtering, and *event aggregation models* for aggregating provenance events.

Provenance Classes. Provenance classes are high-level abstractions for system-level provenance relationships. URSPRUNG allows users to select different provenance classes and underneath configures the corresponding system call filters. Those filters are then pushed down to each node in the cluster.

Currently, URSPRUNG supports four provenance classes: (i) file-based provenance, which captures the interactions between processes and files; (ii) process-group provenance, which captures information on Linux process groups and displays process group membership; (iii) pipe-based inter-process communication (IPC), which tracks pipes between processes to capture IPC; and (iv) network provenance, which tracks creation of sockets and connections to capture remote process interactions. Other classes such as IPC through shared memory can be added.

Due to their abstraction level, provenance classes make it easy for non-expert users, such as data scientists, to configure what provenance should be collected. Instead of requiring knowledge on what system calls constitute what type of provenance relationship, users can select what relationships they want to capture (e.g., files, processes, network) by simply enabling/disabling the corresponding provenance classes through a checkbox in the GUI. By default, all classes are enabled in URSPRUNG.

Event Aggregation Models. An event aggregation model defines how to aggregate lower-level events into semantically richer ones. Such models can be applied to raw events, and can also raise previously aggregated provenance events to higher semantic levels. Aggregation reduces the amount of collected provenance data, and also curates events into more abstract entities to simplify querying.

One example for such an aggregation is the creation of process events. Our data scientist colleagues do not care about `fork`, `execve`, and `exit` system calls, but they do care about the arguments given to their Python processes. So URSPRUNG aggregates raw system call events from `auditd` for `fork`, `execve`, and `exit` system calls.[3] URSPRUNG extracts relevant information from each of these events, e.g., process start time (from `fork`), process command line (from `execve`), and process end time (from `exit`). URSPRUNG's provenance daemons assemble each process event gradually. Once complete, the event is emitted for storage in the provenance database. As an application-level example, Spark jobs can be assembled from two different log entries, one entry signaling the start of the job and one signaling its completion.

Event aggregation is stateful, as the curated event must be accumulated from a series of incoming events. If the cluster is unstable or delays affect time-to-insight for users, it is possible to already emit partial events to the provenance store to persist the state and then augment those events later.

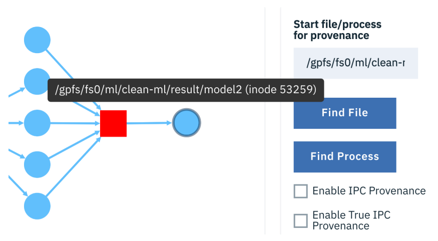
4. THE URSPRUNG GUI

Next, we introduce the URSPRUNG GUI. We first describe how users can navigate and explore the provenance graph (§4.1), and then discuss URSPRUNG's differential provenance views (§4.2) and reproducibility support (§4.3).

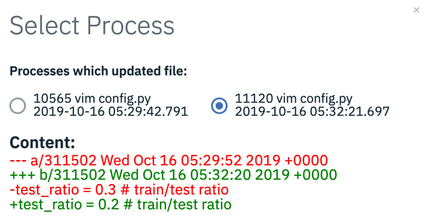
4.1 Graph Exploration

Visualizing provenance graphs for users to consume has been a difficult problem due to the mostly large scale of such graphs [49]. URSPRUNG's graph explorer view (see Figure 2a for a screenshot)

[3]This also includes calls with similar functionality such as `clone`, `vfork`, and `exit_group`.


/gpfs/fs0/ml/clean-ml/result/model2 (inode 53259)
Start file/process for provenance
/gpfs/fs0/ml/clean-r
Find File
Find Process
Enable IPC Provenance
Enable True IPC Provenance

(a) Graph Explorer

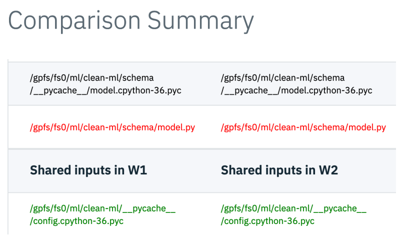


○ 10565 vim config.py 2019-10-16 05:29:42.791
◉ 11120 vim config.py 2019-10-16 05:32:21.697

Content:

```
--- a/311502 Wed Oct 16 05:29:52 2019 +0000
+++ b/311502 Wed Oct 16 05:32:20 2019 +0000
-test_ratio = 0.3 # train/test ratio
+test_ratio = 0.2 # train/test ratio
```

(b) Content Viewer



/gpfs/fs0/ml/clean-ml/schema /__pycache__/model.cpython-36.pyc	/gpfs/fs0/ml/clean-ml/schema /__pycache__/model.cpython-36.pyc
/gpfs/fs0/ml/clean-ml/schema/model.py	/gpfs/fs0/ml/clean-ml/schema/model.py
Shared inputs in W1	**Shared inputs in W2**
/gpfs/fs0/ml/clean-ml/__pycache__ /config.cpython-36.pyc	/gpfs/fs0/ml/clean-ml/__pycache__ /config.cpython-36.pyc

(c) Differential View

Figure 2: URSPRUNG GUI features

has several features which aim to reduce complexity for users during exploration by trying to keep the graph clear.

To display the provenance graph, URSPRUNG uses a node-link diagram visualization. Users can input an object of interest into a search field and then retrieve the provenance for it. However, instead of fetching the entire graph at once, URSPRUNG only retrieves the initial node. Starting from this node, users can then interactively explore the graph by clicking on the node and expanding its immediate dependencies (1-hop neighbors). Input dependencies will be placed on the left of the node while output dependencies appear on its right. This allows users to analyze each step in detail and selectively follow new dependencies at their own pace.

Besides step-by-step exploration, URSPRUNG also uses several other techniques to improve the graph visualization. First, it uses shapes and colors to distinguish different types of nodes and different relationships. For example, processes are displayed as red squares while files are blue circles; orange edges denote file renames, and blue edges indicate read/write dependencies. Second, node labels, such as file names, are displayed in tool tips. They only appear when hovering over a node, decluttering the display. Third, nodes can be manually moved and arranged by users to change the layout of the graph. Finally, users can define filters to remove nodes not relevant to the task at hand.

If the content of a file is tracked (§3.2), URSPRUNG can display content changes as part of the provenance graph (see Figure 2b). A right click on a specific file will open a dialog box, which lists all the different versions of a file with the process that changed it and the time of the change. Users can select one of the versions and a diff between the selected and the previous version is displayed. This can help, e.g., to analyze how a parameter changed between two runs of a data science pipeline.

4.2 Differential Summary Views

As the provenance graph can get difficult to explore the larger it gets, URSPRUNG allows users to compare different pipeline executions through a *differential provenance* summary view. As stated by recent work [27], analyzing the differences between two runs rather than looking at the provenance for a single run can help users better understand why a run produced a particular result.

To use the differential provenance view, users must select two executions of a pipeline for comparison. URSPRUNG will then compute the dependencies for each of the runs and display the differential summary (see Figure 2c). Currently, the summary shows which files were used by both runs, which files were specific to each run, and what output was produced. We are planning to extend this view with a richer comparison to meet the needs of our users.

URSPRUNG uses three criteria to identify different files: the file path, the inode, and the file *version*. The file path and inode are provided by the base provenance sources. URSPRUNG calculates file versioning itself using a hash over its contents. Each time a

Algorithm 1 Constrained prioritized BFS to compute the set R of input dependencies for node v_s

```
Require: $G = (V, E)$, $\forall v \in V : v = (h, t)$, $\forall e \in E : e = (u, v, t)$, $v_s$
 1: $R = \varnothing$, $PQ = \varnothing$, $v_s.t = \infty$, $\forall v \in V : v.h = 0, v.t = 0$
 2: $PQ$.enqueue($v_s, v_s.t$)
 3: while $PQ \neq \varnothing$ do
 4:     $v = PQ$.dequeue()
 5:     if $v \notin R$ then
 6:         $R = R \cup \{v\}$
 7:         $U = \{u \in V | (u, v, t) \in E \wedge t \leq v.t\}$
 8:         $\forall u \in U : u.t = e.t$ where $e = (u, v, t)$
 9:         if isFile($v$) then
10:             $u_{\max} = \max_{u \in U}(u.t)$
11:             $v.h = $ getVersion($u_{\max}.t$)
12:         $\forall u \in U : PQ$.enqueue($u, u.t$)
13: return $R$
```

file modification is detected, URSPRUNG will compute the file hash and store it with the modification event. If not required, version tracking can be disabled to reduce overhead.

To compute the differential summary, URSPRUNG must compute the dependencies for each individual run of the pipelines that should be compared. URSPRUNG does this using a modified breadth-first-search (BFS) as shown in Algorithm 1. The main modifications are the use of a priority queue for ordering, and a constrained traversal of edges to observe time boundaries.

The BFS modifications ensure two properties: (i) dependencies, such as processes modifying files, which occurred after the run of the pipeline are excluded from the pipeline's dependencies. This is achieved through the constrained neighbor detection; (ii) all dependencies that occurred before the run of the pipeline are included. This is achieved through the use of a priority queue, which ensures that the timestamp of a newly detected node is set to the timestamp of its latest incoming dependency. As a result, the maximum subset of neighbors is detected for that node.

The algorithm takes as input the complete provenance graph and a start node v_s (see Algorithm 1). Edges are directed and labeled with a timestamp t, which indicates, when the dependency occurred. Each node in the graph has a version h and a timestamp t, which are initially set to 0, except for v_s (Line 1). The priority queue PQ is initialized with v_s and its corresponding timestamp as the priority (Line 2). The BFS then starts by removing the first element from the queue and retrieving all of its ancestor nodes (Lines 3–7). The ancestor retrieval is constrained to only nodes whose edges have a timestamp less than the current node (Line 7). Once found, the timestamp of each ancestor is set to its corresponding edge timestamp (Line 8). If the current visited node is a file node, it also needs to have its correct version set, i.e., the version corresponding to the last edit of the file (Lines 9–11), to ensure that the correct versions are compared in the differential view.

4.3 Reproducibility Support

URSPRUNG helps to improve both *repeatability* and *reproducibility*. By repeatability, we mean activity that does not actively reproduce a result but helps users understand how a result was created so they are able to recall and repeat the steps. By reproducibility, we mean supporting users to actively re-run the computational steps necessary to recreate a result. These definitions are similar to ACM's terminology [11] but are not standardized [64].

Repeatability is supported in URSPRUNG through the provenance graph explorer and the additional features presented above. Through the provenance graph, users can explore a data science pipeline by investigating the associated processes and their parameters. They can drill down into the different pipeline steps by analyzing, how the content of code or configuration files has changed and compare entire pipelines through the differential view.

To actively reproduce a result, URSPRUNG can generate the list of processes that need to be rerun to regenerate the result file. Using the GUI, users can "long click" on a node in the graph to get its associated list of generative processes. The list is ordered and contains the exact commands for each step of the pipeline, including the machine on which it was run. Additionally, if application-specific provenance is collected, URSPRUNG can detect whether a result was created by a higher-level job. URSPRUNG will then display only the information necessary to launch the job, not to run each individual process therein. Finally, URSPRUNG's version tracking can be used to notify users if dependent files have changed since the desired result file was created. If content tracking is enabled for the changed files, URSPRUNG can assist in reverting them to the appropriate versions.

5. IMPLEMENTATION

We implemented the URSPRUNG collection system in C++ and the GUI with Node.js. URSPRUNG currently relies on `auditd` and IBM Spectrum Scale's Watch Folder as system-level provenance sources. The provenance store is a Db2 database. In the implementation, we address the challenges of reliable (§5.1) and timely (§5.2) event delivery. We also discuss scalability considerations (§5.3) and describe current limitations of our prototype §5.4.

5.1 Reliable Event Delivery

Reliable event delivery is crucial for provenance collection. Missing an event means missing relationships, affecting the precision of the provenance graph. This applies in particular to system-level provenance as it forms the basic relationships between entities.

We analyze URSPRUNG's reliability properties in terms of the event pipeline. We assume all events are persistent once they reach the distributed message queue. Application-level events are retrieved by the rule engine *downstream* from the persistent message queue, and are thus likewise reliable. Hence, the main source of unreliability comes from the *upstream* event sources. URSPRUNG depends on two upstream sources for base provenance: Watch Folders for file system events, and `auditd` for other system events.

Watch Folders is Spectrum Scale's event notification mechanism. Its API permits users to subscribe to cluster-wide file system events such as the opening, closing, or creation of a file. The choice to use Watch Folders notifications has the main benefit of obtaining richer metadata for a single event. In particular, Watch Folders notifications include how many bytes a process has read from/written to a file. This allows URSPRUNG to reliably determine read/write relationships without having to track individual `read` and `write` calls, which significantly reduces overhead. Watch Folders is reliable; it guarantees event delivery in most cases. The only exceptions are node crashes, network failures, and file system panics. However, such cases are rare and would disrupt provenance collection in any case, so we view them as an acceptable risk.

In contrast to Watch Folders, `auditd` is a best-effort system. We configured `auditd` with blocking and unbounded queues, but there is a bounded retry limit when adding events to the `auditd` plugin queue[4] after which vanilla `auditd` will give up when under heavy load. We extended `auditd` to permit infinite retries, giving users a trade-off between system performance and precision in the provenance graph. Despite these modifications, we still observe occasional `auditd` event loss during heavy bursts. We are investigating this behavior to explore whether it is possible to guarantee event delivery in `auditd` and the implication of such guarantees.

5.2 Timely Event Delivery

In addition to reliably capturing provenance events, these events must also move quickly through the pipeline to capture transient provenance data (§3.2). If events are processed too late, the information becomes stale and the rule will not collect any data. For example, if a rule exists to extract provenance from the `stdout` of a process, the capture should start promptly after the `execve` call is issued. As another example, to extract data from a temporary file, URSPRUNG must start monitoring before the application deletes it.

URSPRUNG keeps event latencies low by minimizing processing on the critical path. Besides some data transformations and (de)serialization, no other synchronous processing occurs. Rule evaluation and content tracking happen asynchronously. Each type of rule has a task queue and one or more worker threads to execute the rules. Whenever an event is received that matches the conditions of a rule, that event will be added to the corresponding task queue and processed asynchronously.

While these strategies reduce the processing latency, our choice of an asynchronous delivery pipeline means that URSPRUNG cannot eliminate latency entirely. Thus URSPRUNG's rule engine may race with short-lived processes or files, leading to potential provenance loss. Recall, however, that we chose the asynchronous design in light of our problem context: data science pipelines. Data science pipelines typically consist of long-running processes and long-lived files, and so URSPRUNG can be applied in those cases without losing provenance.

5.3 Scalability

Several aspects of URSPRUNG's design and implementation facilitate scalability. Each of its components (event queue, collection system, provenance store) can be scaled to support larger analytics clusters and bursty workloads. Following Figure 1, ordered from provenance generation to storage:

Event Queue. URSPRUNG uses Kafka as its event queue. Kafka can be scaled with the cluster size by adding additional message brokers on different nodes, which ensures fast event delivery to the collection system.

Collection System. To ensure low latencies and high event throughput, even under high loads, the collection system itself can be scaled out. First, URSPRUNG leverages Kafka's consumer groups to partition event streams from provenance sources and processes individual partitions in parallel. The partition keys are chosen such that

[4]URSPRUNG uses an `auditd` plugin to transfer `auditd` events to its Kafka queue.

Table 2: Raw event statistics for application use cases

Workload	#events	event rate	event size
CleanML	7.2 K	75/s	1.3 MB
ImageML	13.2 K	28/s	2.6 MB
Spark	0.6 M	1,400/s	92 MB
Vanderbilt	3.2 M	13,900/s	540 MB

consumers do not share state across partitions and hence, the collection system is distributed. Second, as mentioned in §5.2, events inside a single consumer are processed in parallel as stateless rules are executed by different threads. Third, event aggregation and filtering is pushed upstream to the provenance sources whenever possible to distribute the computational load across all cluster nodes.

Provenance Store. To prevent the ingest into the provenance store from becoming a bottleneck, URSPRUNG uses a scalable storage solution (Db2 in our case). Additionally, URSPRUNG supports asynchronous loading of provenance data into the provenance store to avoid congestion during peak load periods (see §6.3).

5.4 Current Limitations

Our current prototype implementation of URSPRUNG still has several limitations. First, as it is currently based on Spectrum Scale Watch Folders, it requires a Spectrum Scale file system on which the data science pipelines are executed. However, the dependency on Watch Folders is not strict and the same functionality can be achieved through other mechanisms [70, 33, 66].

Another limitation is that URSPRUNG is currently not tracking software and environment dependencies such as library or operating system versions for pipeline executions. We plan to add support for environment tracking in URSPRUNG through techniques such as tracking `mmap` calls (for dependencies) and capturing the process environment (e.g., by calling `env` or `pip freeze`) whenever a certain process is executed.

6. EVALUATION

In our evaluation, we aim to understand if URSPRUNG is able to capture the necessary provenance for different data science pipelines (§6.2). We also study URSPRUNG's performance in terms of how much overhead it adds and under varying event loads (§6.3), the impact of provenance classes on storage and processing overhead (§6.4), the performance of rule executions (§6.5), and URSPRUNG's ability to scale in larger analytics clusters (§6.6).

6.1 Experimental Setup

Our testbed consists of a 13 node cluster, 12 nodes for running the workloads and one node for URSPRUNG. Each node has an Intel Xeon Gold 6142 CPU with 64 cores running at 2.6 GHz and 128 GB memory. Nodes are connected through 100 Gbps Ethernet and storage is provided from an IBM ESS 3000-based system, consisting of 24 NVMe devices and connected through 3x100 Gbps Infiniband to the nodes. In our charts, we plot the median measurement and indicate the 25th and 75th percentiles as error bars.

Application Workloads. We use four main workloads to evaluate URSPRUNG: CleanML, Vanderbilt, Spark, and ImageML. Vanderbilt and ImageML are real-world examples, used by colleagues at Vanderbilt University and IBM Almaden. CleanML and Spark are benchmarks resembling common machine learning workloads.

The **CleanML** workload is taken from the recently published CleanML benchmark [46] and models the steps of a common machine learning pipeline. The steps include initialization to split the input data into train and test set, cleaning to remove missing values, preprocessing to extract labels, and finally training. We use the provided Airbnb dataset and train a logistic regression model for predicting Airbnb property prices. The pipeline is python-based and we modify the code such that each step is run separately. The raw Airbnb input data is 13 MB.

The **Vanderbilt** workload is an image analysis pipeline that processes cell images from a High Throughput Screening device for drug analysis used by researchers at Vanderbilt University. The pipeline first analyzes image metadata to create job descriptions for image pairs and then submits each job to the IBM® Spectrum® LSF® scheduler [2] for parallel processing. Once the processing has finished, a post processing step is run to produce a result summary. The pipeline consists of R and python scripts and is programmed using the common workflow language (CWL) [1]. The total input data set is 16 GB.

The **Spark** workload is an example workflow from Databricks, which demonstrates Spark machine learning pipelines [29]. The pipeline trains a logistic regression model to predict whether a customer's product review is good or bad. It first preprocesses the text data by bucketizing and tokenizing the individual reviews and then creates a vectorized representation which is used as an input to training. The job is written in pyspark and uses the Amazon review dataset as its input. The dataset size is 55 GB.

The **ImageML** workload is a machine learning pipeline used by IBM Research to classify X-ray images. It includes model training and deployment. In the training step, a TensorFlow-based deep neural network is trained for image classification. The resulting model is deployed as a REST service so users can `POST` images for inference. The pipeline is a mix of python and bash and the training step accesses 1,174 images. As part of our pipeline execution, we send one inference query to the REST service.

Table 2 shows the raw event statistics for each workload, i.e., how many raw events are generated in the provenance sources and at what rate. The size is determined by the overall disk space the events consume when represented as an uncompressed `.csv` file. The CleanML and ImageML workloads are very light and only generate several thousand raw events, occupying few MB at a rate of tens of events per second. The Spark workload is an intermediate workload with hundreds of thousands of events at a rate of roughly 1,400 events/second. The Vanderbilt workload is the most demanding workload with millions of events being generated for a total amount of 540 MB and at a rate of almost 14,000 events/second.

6.2 Usability

To evaluate URSPRUNG's utility, we compile a set of common usability requirements for data science reproducibility. These requirements are taken from a variety of sources [73, 42, 58, 24, 69] and from conversations with IBM Research colleagues who work on machine learning. We extract six requirements that provenance should address in this context:

R1) It must track changes to the dataset.
R2) It must track changes to algorithms, code, and parameters.
R3) It must track pipeline metadata during execution such as partially trained weights, ran epochs, etc.
R4) It must track framework dependencies.
R5) It must support different tools and frameworks.
R6) It must require minimal effort from users to get benefit.

SPADE. We compare URSPRUNG to SPADE [33]. SPADE is a popular open-source provenance collection system. In the literature, we believe it is the closest to URSPRUNG in design and functionality. It supports combining operating system provenance from

Table 3: Comparison of URSPRUNG and SPADE in terms of usability requirements (parentheses indicate requirements that are feasible but not yet supported in the prototype)

	R1	R2	R3	R4	R5	R6
Ursprung	✓	✓	✓	(✓)	✓	✓
SPADE	✓	—	—	✓	✓	✗

`auditd` with higher-level application-specific provenance. For system provenance, SPADE's users can enable/disable a certain set of system call events in `auditd`. We believe URSPRUNG's concept of provenance classes provide simpler, more fine-grained control (§3.3). For application-specific provenance, SPADE requires users to manually send application-specific provenance to SPADE's collection server, e.g. through code instrumentation, implementing a *reporter*, or LLVM-based code annotation. We believe URSPRUNG's application-level language for provenance capture (§3.2) also simplifies the capture of application-specific provenance.

Requirements Analysis. Table 3 summarizes which requirements are addressed by URSPRUNG and SPADE. We will discuss each requirement in detail in the following.

URSPRUNG and SPADE both satisfy R1 through base provenance collection. Base provenance captures all changes that are made to a dataset including addition, removal, and updating of items. This allows both systems to capture updates in terms of data versions and the process, which updated the data. URSPRUNG additionally allows to record the actual changes through a Track rule. However, this may be infeasible for larger data sets.

R2 is supported by URSPRUNG through Track rules, which can track file content and hence capture how specific code or configuration parameters have changed between different executions of a pipeline. While this is technically possible in SPADE, e.g., through implementing a custom reporter, which watches the necessary files and manages their content, it requires high implementation effort from the user (indicated by a dash in Table 3).

Information regarding R3 is usually outputted in either log files, databases, or to the command line. All of these sources can be accessed through URSPRUNG's LogTransfer, DBTransfer, and CaptureCout rules while larger temporary data can be indexed through FileLoad rules. Similar to R2, capturing this information in SPADE would only be possible with significant effort for the user.

R4 is enabled in URSPRUNG and SPADE through base provenance collection. URSPRUNG currently only supports dependency tracking if processes access dependencies (e.g., libraries) through standard file IO as we have not yet implemented support for other loading mechanisms, e.g., through `mmap`. SPADE can track dependencies for both approaches.

R5 is met by both URSPRUNG and SPADE as they sit at a layer below applications and hence, support provenance capture for any tool or framework that is run on top.

Overall, we conclude that R6 is fulfilled by URSPRUNG by design. URSPRUNG's main goals are transparency and simplicity, and we strive to minimize the configuration options exposed to users. URSPRUNG minimizes the impact on data scientists because it requires neither changes to existing workflows, nor a deep understanding of the applications. As SPADE does not support R2 and R3 out of the box, R6 is not fulfilled as users would need to spend significant effort to collect application-specific information, relevant to different data science pipelines.

Case Study. As a case study for its usability, we run the four application workloads and configure URSPRUNG to collect all necessary provenance. We are able to do so using base provenance plus a small set of 11 rules. For the Vanderbilt workload, we collect data from LSF and CWL logs to list different executions of the pipeline and find the output data of a specific run. Additionally, we also load temporary data created as part of the workload into the database. This requires a total of 6 rules (3 FileLoad, 2 DBTransfer, 1 LogTransfer). For the CleanML workload, we use 1 Track rule to obtain parameter and algorithm changes, by monitoring changes to all python files in the workload's root directory. For the ImageML workload, job information like the current epoch and accuracy is printed to stdout. To capture this information, we use 1 CaptureCout rule. For the Spark workload, we use 3 LogTansfer rules to capture job submission information and all warning and error logs.

6.3 Performance

Next, we study URSPRUNG's performance in terms of its overhead on application workloads and the event rates it can sustain in terms of latency and throughput. For the overhead analysis, we use the four application workloads. To study latency and throughput, we use a set of four microbenchmarks, each of them repeatedly producing the same set of events.

The four microbenchmarks are: 1) open-close, which opens and closes a file; 2) fork-exec-exit, which runs a new process; 3) pipe-dup-close, which creates two processes and establishes a pipe between them; and 4) socket-connect, which creates a process that opens a TCP socket to a server. To stress the system, we also push raw events into the provenance database, besides any aggregated provenance events. This allows us to increase the event load on URSPRUNG. We run each benchmark for 20 s and measure event latency and overall throughput. As we are limited to a single database node in our experiments, we write incoming events to disk first and asynchronously import them into the database to prevent the database ingest from becoming a bottleneck. This does not impact reliability as events are persisted to disk.

Overhead. To measure URSPRUNG's impact on performance, we run the four application workloads and compare completion times to the baseline (without any provenance collection) and SPADE. Each run is repeated five times, except the CleanML workload, which is repeated 20 times as its completion time was more variable. Both the CleanML and ImageML workloads are run on a single node while the Vanderbilt and Spark workloads are distributed and run on five nodes (we scale out the cluster in §6.6).

We configure SPADE to produce the least amount of provenance and make it comparable to URSPRUNG. We disable tracking for memory- and IO-related system calls, use a blacklist filter to only process events from the Spectrum Scale file system, and apply a deduplication filter to remove duplicate edges and vertices. We also found it necessary to modify SPADE's implementation, giving `auditd` an unlimited event buffer in the kernel. Otherwise, the Vanderbilt workload exhausts the buffer and the machine stalls.[5]

Overall, the results show that the overhead is small for both URSPRUNG and our modified version of SPADE for all workloads (see Figure 3). CleanML has the highest overhead (8.5% and 10% for URSPRUNG and SPADE, respectively) in the median. However, CleanML is noisy in general due to a non-deterministic training phase, and the completion times for all three setups are within the error bars of each other. Both ImageML and Spark experience no visible overhead with provenance collection enabled.

The largest consistently observable overhead is incurred by the Vanderbilt workload. We observe a 4% overhead added by URSPRUNG and a 6.5% overhead added by SPADE in the median.

[5]We attribute this to excessive error handling/logging in the kernel.

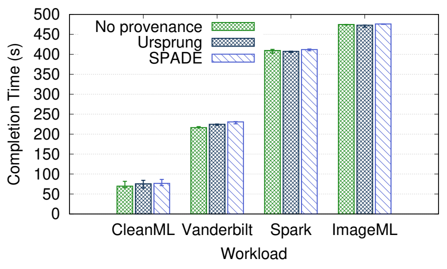


Figure 3: Workload completion times

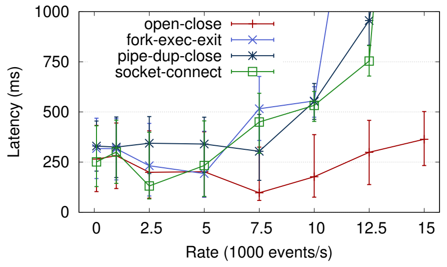


Figure 4: Latency for different event rates

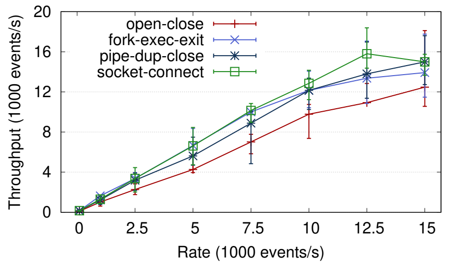


Figure 5: Throughput for different event rates

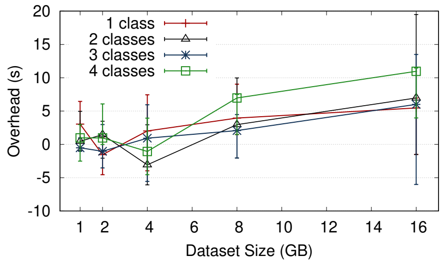


Figure 6: Overhead for provenance classes

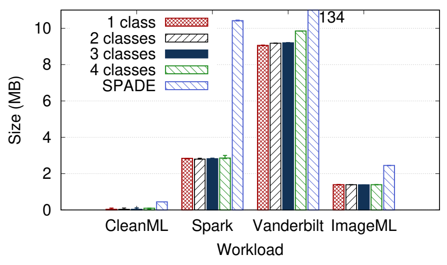


Figure 7: Provenance storage demand

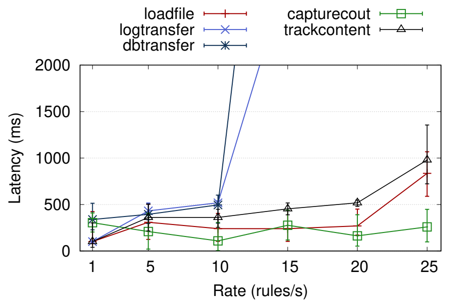


Figure 8: Rule execution latency

Unsurprisingly, Vanderbilt is also the most provenance-intensive workload. It launches approximately 900 LSF jobs, which each access two small input files and write three output files. This results in a larger rate of events compared to the other workloads (see Table 2), which spend most of their time in compute and large file IO and hence, incur a smaller number of system calls. As SPADE is tracking more system calls compared to URSPRUNG, which cannot be disabled, SPADE adds an additional overhead of 2.5%.

As a result, we see that for most machine learning pipelines, which consist of long-running steps and large file IO, URSPRUNG does not add any significant overhead. For workloads that consist of a large number of smaller steps, URSPRUNG's overhead is visible but remains acceptable within 4%. The finer-grained control over system calls through provenance classes gives URSPRUNG an advantage over existing systems under high load.

Latency. Looking at the latency results (see Figure 4) shows that URSPRUNG is able to keep latencies stable as event rates increase. Overall, latencies stay below 500 ms up to an event rate of 10,000 events per second. The open-close microbenchmark shows the lowest latencies, which are stable around 150 ms. This is due the fact that the corresponding events, triggered by the file close, are generated by Watch Folders and are directly pushed to Kafka, whereas events from the other microbenchmarks arrive via `auditd` and go through an event source model for aggregation. This leads to slightly higher latencies for events from the `auditd` source. For event rates over 10,000, events start to queue and latencies rise.

Throughput. The throughput for the microbenchmarks is shown in Figure 5. Overall, it increases linearly with increasing event rates, which means that URSPRUNG can scale to higher event rates. The maximum sustained throughput is around 15,000 events/second, apparently the maximum throughput that a single Kafka consumer in URSPRUNG can sustain. We also see that the `auditd`-based microbenchmarks show higher throughputs than the actual event rates. This is due to *event amplification*, i.e., after raw events are aggregated to provenance events, both raw and provenance events are delivered, increasing the total number of events.

6.4 Provenance Classes

We now study the impact of provenance classes on processing and storage overhead. We aim to answer the question of how much overhead is added by enabling additional provenance classes. We start from the base class of file-process interactions (1 class) and then gradually add process group tracking (2 classes), pipe-based IPC (3 classes), and network provenance (4 classes).

Processing Impact. To analyze the impact of provenance classes on processing overhead, we use the Vanderbilt workload as it is the most provenance-intensive. We vary the input dataset size from 1 to 16 GB and measure completion times for the different provenance class configurations compared to a run without URSPRUNG. We repeat each experiment five times and plot the overhead in Figure 6.

For smaller dataset sizes, there is no observable difference between the different provenance class configurations. This is because the workload only runs for a short duration and hence, the overall completion time variation dominates any potential benefits.

As the input dataset grows to 8 GB and 16 GB, URSPRUNG's overhead becomes more visible. Specifically, for all four classes enabled, the median overhead increases by approximately 5 s compared to the other configurations. This translates to an additional overhead of 2.5%. We do not observe a significant difference between three or less classes. This is due to the fact that the Vanderbilt workload does not generate a large amount of events in those classes and hence, their impact is not visible. Only once all classes are enabled, the overhead increases and becomes significant.

The results yield two key observations: 1) Provenance classes are able to reduce overhead; 2) The reduction depends on the load generated by each class. For example the Vanderbilt workload does not create any pipes and hence, enabling pipe-based IPC tracking does not make any difference. The heavier the load in each class, the more overhead can be reduced by disabling that class.

Storage Impact. To evaluate the potential storage benefits of tunable provenance classes, for each of the workloads we measure the amount of provenance data generated under the same provenance class configurations as in the previous experiment. We also com-

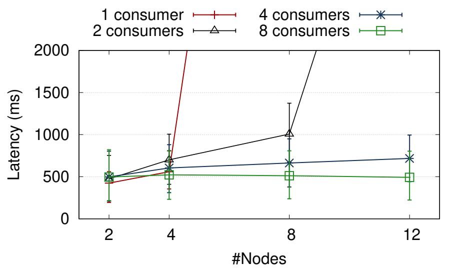


Figure 9: Latency as a function of cluster size

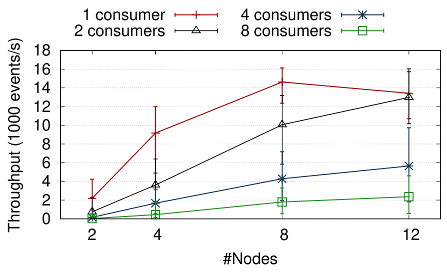


Figure 10: Throughput *per consumer* for increasing cluster size

pare URSPRUNG's storage demand to SPADE. We repeat each measurement five times to cover non-determinism in the workloads and the underlying system. The results are shown in Figure 7.

For the CleanML, Spark, and ImageML workloads, we do not observe significant difference between the provenance class configurations. This is for the same reason described above: these workloads do not generate any significant load in the higher provenance classes (e.g., they do not use pipes or socket IO), so enabling those provenance classes does not impact storage consumption.

For the Vanderbilt workload, we observe a slight increase of 0.12 MB between 1 and 2 classes and of 0.65 MB between 3 and 4 classes. This results in an 8% overall storage saving when comparing 1 and 4 classes. The storage space reduction will vary based on the load generated by each provenance class. If the load is higher, the space savings will increase further.

Compared to SPADE, URSPRUNG requires significantly less storage across all 4 workloads (ranging from $1.8\times$ for ImageML to $14.6\times$ for Vanderbilt). Upon investigation, we found two reasons for this gap. First, URSPRUNG is tailored to provenance for the data science use case and hence, collects less data in general. Second, URSPRUNG's event aggregation models curate raw events and store only the relevant information into the database, which further reduces storage consumption.

6.5 Rule Execution

Next, we study the performance of rule executions. As mentioned in §5.2, the latency between an event that triggers a rule and the actual rule execution needs to be low to ensure that transient provenance can be captured. Rule execution happens asynchronously in separate threads and events that trigger a rule are queued until a processing thread becomes available. In this experiment, we are interested in the maximum rate at which rules can be processed before this event queue starts to build up.

Our workload is a set of microbenchmarks that allows us to trigger repeated execution of one of the five rule types at a specified rate. We measure the latency between an event's arrival in the message queue and the execution of the rule that it triggers. We vary the rate of event arrival and plot the results in Figure 8.

Our first observation is that the overall rate at which rules can be processed is low (tens of rules per second). Rule execution is expensive! For example, a DBTransfer rule needs to establish a connection to an application-specific database, submit a SQL query to extract the latest provenance records, and insert this information into the provenance database. Hence, for LogTransfer and DBTransfer rules, the maximum sustainable rate is 10 rules/second.

For the CaptureCout and FileLoad rules, URSPRUNG is able to keep latencies stable for higher rates as these rules are *stateless*, i.e., they do not have to keep track of file or database offsets. Hence, new incoming rules can be processed in parallel. While the TrackContent rule is stateful, it is still cheaper to process compared to the DBTransfer and LogTransfer rules as it does not have to parse file/database contents but only copies and commits a file to a local repository. In our benchmark, the tracked files were small (bytes) but the cost of TrackContent rules increases with larger files.

Overall, the results show that URSPRUNG rules are suitable for infrequent events (tens per second). If events are more frequent, timely event delivery cannot be guaranteed and transient provenance records may be lost.

6.6 Scalability

Finally, we investigate URSPRUNG's capability to scale to larger clusters. We use our heaviest provenance workload, Vanderbilt, and run it on clusters of 2, 4, 8, and 12 nodes. For each configuration, we have one dedicated LSF master node, which manages job submissions but does not run jobs itself. We use the rest of the cluster as workers. We use Kafka consumer groups to scale out URSPRUNG's consumers to parallelize event processing and improve throughput. We measure event latency and throughput for different numbers of consumers and, similar to §6.3, we enable raw events to be processed by URSPRUNG to increase the load on the system.

In our initial experiments, we found that by default, the Vanderbilt workload submitted all its corresponding jobs to LSF at the start of the workload. This generated a high volume of concurrent events on the master node (hundreds of thousands), which led to long event queues in the `auditd` provenance source and resulted in high latencies. To mitigate this effect, we changed the workload such that jobs are submitted gradually as old jobs finish to distribute the load on the master node over the entire workload duration.

Latency. The latency results are shown in Figure 9. For smaller clusters, the overall event load is lower as there are less nodes generating events. Hence, URSPRUNG is able to achieve stable latencies in the range of 500 ms with a lower number of consumers and even a single consumer is sufficient. As the cluster size increase, the event load increases. In the case of a single consumer, a size of 8 nodes leads to latency degradation, while 2 consumers are not able to sustain the rates in a 12 node cluster. The 4 and 8 consumer setups are able to keep latencies stable for all cluster sizes.

Throughput. The throughput results in Figure 10 show a similar trend as the latency results (note that the throughput is shown per consumer so the overall system throughput for N consumers is N times the reported value). A single consumer cannot sustain the event rate when the cluster size reaches 8 nodes and caps at around 14,000 events/second. With more consumers, the individual throughput rates decrease but the overall system throughput increases, reaching a maximum of 24,000 events/second for 12 nodes and 4 or 8 parallel consumers.

In summary, this shows that URSPRUNG is able to scale with the size of the cluster through parallelizing event consumption. However, if a large burst of events is generated in a provenance source, latencies will deteriorate until URSPRUNG is able to process all outstanding events and recovers.

7. RELATED WORK

Provenance Capture Systems. Research has proposed a variety of provenance collection systems in the past. One of the earliest systems is PASS [56], which proposes to capture provenance transparently in the kernel through system call interception. However, PASS is not able to collect application-specific provenance. Later systems such as PASSv2 [55], Story Book [68], and the CPL [50] allow to augment system-level provenance with application-specific provenance. However, these systems require to change the application by adding additional calls to emit provenance records, which is infeasible for the variety of tools used in data science.

Ghoshal et. al. [34] proposed the idea of capturing provenance from log files. Similar to URSPRUNG, they propose a rule language to define capture rules to extract provenance from log files. However, they do not capture system-level provenance and also do not support other sources such as databases or process `stdout`.

Like URSPRUNG, SPADE [33] and CamFlow [60], are provenance collection systems that support both transparent system-level and application-specific capture. However, we believe URSPRUNG is easier to configure, and integrates a rule-based capture system rather than requiring users to know low-level information, e.g., an application's function call pattern.

Machine Learning Model Management. Kumar et. al. [44] introduced the idea of model selection management systems. Such systems structure the model creation process and are also able to collect the provenance of different models. This vision has been implemented by a variety of research systems such as ModelDB [71], ProvDB [53], Schelter et. al. [67], and Sridhar et. al. [69]. There is also a set of open source projects to help with managing the model lifecycle, e.g., MLflow [8], CodaLab [5], and Pachyderm [9].

However, the above solutions are either not fully transparent, i.e., they require code annotations or the use of a specific tool to run and/or define pipelines [69, 8, 5, 9], or only support a fixed set of tools [71, 53, 67]. Compared to that, URSPRUNG aims at full tool flexibility and generality. While this can sometimes lead to less detailed provenance, it still suffices for the majority of uses cases.

Big Data Provenance. Adding provenance capture to big data processing systems has been another field of active research. Systems such as Lipstick [16], RAMP [59], and HadoopProv [14] explore provenance capture for MapReduce while Titian [41] is targeted at Spark. Newt [47] is instrumentation-based and hence, can support different platforms. As the above systems are built for specific applications, they are able to collect detailed, record-level provenance for jobs and pipelines. While this provides finer granularity, it is harder to generalize beyond the big data domain.

Workflow Management. Scientific workflow management systems such as VisTrails [25], Kepler [48], or Taverna [74] allow scientists to define workflows and capture their provenance during execution. While the collected provenance is detailed and allows for easy reproducibility, the problem is that users need to define and run their workflows through these specific systems, limiting their flexibility and adding additional effort.

noWorkflow [57, 63] allows to capture provenance from python scripts, including script-specific information such as declared functions and information on the execution environment. While this works transparently and provides fine-grained, application-specific provenance relationships, e.g., how data was transformed by individual function calls inside the script, it is harder to generalize to a broader set of non-python based applications.

Provenance Consumption. The need for consumable provenance for non-expert users has been pointed out by Deutch et. al. [30]. One promising approach in this direction is differential provenance, which was introduced by Chen et. al. for failure root cause analysis [27]. noWorkflow [57, 62] has similarly proposed the *diff-based* analysis of different workflow executions. URSPRUNG's differential view follows noWorkflow's file access diff, which includes files that are added, removed, and changed [62].

Similar to URSPRUNG, SPADE allows temporal traversal of the provenance graph through a *query transfomer* [12]. However, it does not support tracking the specific versions of the nodes in the temporally constrained graph. This behavior, which URSPRUNG supports, is needed to compare two runs of the same pipeline.

Other work has focused on making the provenance graph easier to navigate. Different layouts such as node-link diagrams [49] and radial visualization [22] have been explored. URSPRUNG is most similar to the Provenance Map Orbiter [49], which uses node-link diagrams to display coarse-grained provenance through summary nodes and allows to zoom into summary nodes to view finer grained information. In comparison, URSPRUNG has additional features such as a file diff viewer and allows step-by-step graph exploration.

Data Discovery. Provenance collection has been identified as an important part in the larger space of data discovery to search for and identify existing data sets and their relationships to each other. Goods [36] and Guider [52] both use provenance for this purpose. Decibel [51] is a system that can track provenance and dependencies between data sets, similar to a source code version control system. Compared to URSPRUNG, these systems focus on coarser provenance dependencies and they do not provide more detailed information about the processing and the executed pipelines.

Record & Replay. To improve reproducibility, previous research has studied *record & replay* systems. Such systems support reproducibility either on a per-process or a per-job basis, by leveraging lower-level primitives including memory accesses [61, 20, 45], copy-on-write checkpointing [31], and monitoring the language runtimes [28, 21]. In contrast, URSPRUNG's design assumes that data science pipelines are sufficiently deterministic that higher-level provenance is adequate to reproduce them. Additionally, these systems do not assist with understanding individual pipeline steps.

8. CONCLUSION

The reproducibility crisis in data science and machine learning requires new tools to ease the task of documenting and tracking complex workflow pipelines. We presented URSPRUNG, a transparent provenance capture system, which is specifically designed for the data science domain. URSPRUNG makes capturing base provenance efficient and easy to configure through provenance classes and event aggregation models. It can augment base provenance with application-specific information, collected through a rule-based language without application-level changes. The integrated GUI has various features to ease the exploration of the provenance graph. We have implemented a prototype and showed that it can collect comprehensive provenance for a variety of data science pipelines with only a small overhead of up to 4%. URSPRUNG supports data scientists by simplifying the tracking of their pipelines and thereby helps to improve result reproducibility.

Acknowledgements. The authors wish to thank the anonymous reviewers, whose comments and advice has helped improve this work, and Alexander Lubbock and Darren Tyson, for the invaluable feedback and discussions on the initial prototype.

9. REFERENCES

[1] Common Workflow Language. https://www.commonwl.org/, 2018.

[2] IBM Spectrum LSF. https://ibm.co/2Lpafez, 2018.

[3] Introduction to Watch Folder. https://ibm.co/2q3QBhF, 2018.

[4] Apache Kafka. https://kafka.apache.org/, 2019.

[5] CodaLab. https://codalab.org/, 2019.

[6] Linux Man Pages – auditd. http://man7.org/linux/man-pages/man8/auditd.8.html, 2019.

[7] Linux Man Pages – inotify. http://man7.org/linux/man-pages/man7/inotify.7.html, 2019.

[8] MLflow. https://mlflow.org/, 2019.

[9] Pachyderm. https://www.pachyderm.io/, 2019.

[10] Python Code Glitch May Have Caused Errors In Over 100 Published Studies. https://bit.ly/38DuUq5, 2019.

[11] Artifact Review and Badging. https://bit.ly/2OdPm8c, 2020.

[12] SPADE GitHub – Available Transformers. https://bit.ly/2zib6LY, 2020.

[13] A. Agrawal, R. Chatterjee, C. Curino, A. Floratou, N. Gowdal, M. Interlandi, A. Jindal, K. Karanasos, S. Krishnan, B. Kroth, J. Leeka, K. Park, H. Patel, O. Poppe, F. Psallidas, R. Ramakrishnan, A. Roy, K. Saur, R. Sen, M. Weimer, T. Wright, and Y. Zhu. Cloudy with High Chance of DBMS: A 10-Year Prediction for Enterprise-Grade ML. In *Proceedings of the 2020 Conference on Innovative Data Systems Research (CIDR'20)*, 2020.

[14] S. Akoush, R. Sohan, and A. Hopper. HadoopProv: Towards Provenance as a First Class Citizen in MapReduce. In *Proceedings of the 5th USENIX Workshop on the Theory and Practice of Provenance (TaPP'13)*, 2013.

[15] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann. Software Engineering for Machine Learning: A Case Study. In *Proceedings of the 41st ACM/IEEE International Conference on Software Engineering (ICSE'19)*, 2019.

[16] Y. Amsterdamer, S. B. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen. Putting Lipstick on Pig: Enabling Database-style Workflow Provenance. *PVLDB*, 5(4):346–357, 2011.

[17] G. Barber. Artificial Intelligence Confronts a 'Reproducibility' Crisis. https://bit.ly/30tEBEk, 2019.

[18] A. Bates, D. J. Tian, K. R. Butler, and T. Moyer. Trustworthy Whole-system Provenance for the Linux Kernel. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security'15)*, 2015.

[19] L. Bernardi, T. Mavridis, and P. Estevez. 150 Successful Machine Learning Models: 6 Lessons Learned at Booking.com. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD'19)*, 2019.

[20] S. Bhansali, W.-K. Chen, S. De Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for Instruction-level Tracing and Analysis of Program Executions. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE'06)*, 2006.

[21] R. L. Bocchino Jr, V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A Type and Effect System for Deterministic Parallel Java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'09)*, 2009.

[22] M. A. Borkin, C. S. Yeh, M. Boyd, P. Macko, K. Z. Gajos, M. Seltzer, and H. Pfister. Evaluation of Filesystem Provenance Visualization Tools. *IEEE Transactions on Visualization and Computer Graphics*, 19(12), 2013.

[23] E. Breck, N. Polyzotis, S. Roy, S. Whang, and M. Zinkevich. Data Validation for Machine Learning. In *Proceedings of the Conference on Systems and Machine Learning (SysML'19)*, 2019.

[24] A. Burt. Is There a 'Right to Explanation' for Machine Learning in the GDPR? https://bit.ly/39OCZZ4, 2017.

[25] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo. VisTrails: Visualization Meets Data Management. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'06)*, 2006.

[26] L. Carata, S. Akoush, N. Balakrishnan, T. Bytheway, R. Sohan, M. Seltzer, and A. Hopper. A Primer on Provenance. *Communications of the ACM*, 57(5), 2014.

[27] A. Chen, Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo. The Good, the Bad, and the Differences: Better Network Diagnostics with Differential Provenance. In *Proceedings of ACM SIGCOMM (SIGCOMM'16)*, 2016.

[28] J.-D. Choi, B. Alpern, T. Ngo, M. Sridharan, and J. Vlissides. A Perturbation-free Replay Platform for Cross-optimized Multithreaded Applications. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS'01)*, 2000.

[29] J. Damji and J. Pohl. Building Complex Data Pipelines with Unified Analytics Platform. https://bit.ly/2V90uIj, 2017.

[30] D. Deutch, N. Frost, and A. Gilad. Provenance for Non-Experts. *IEEE Data Engineering Bulletin*, 41(1), 2018.

[31] S. I. Feldman and C. B. Brown. Igor: A System for Program Debugging via Reversible Execution. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, 1988.

[32] J. Freire and F. Chirigati. Provenance and the Different Flavors of Computational Reproducibility. *IEEE Data Engineering Bulletin*, 41(1), 2018.

[33] A. Gehani and D. Tariq. SPADE: Support for Provenance Auditing in Distributed Environments. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference (Middleware'12)*, 2012.

[34] D. Ghoshal and B. Plale. Provenance from Log Files: A BigData Problem. In *Proceedings of the EDBT/ICDT Workshops*, 2013.

[35] M. Haldar, M. Abdool, P. Ramanathan, T. Xu, S. Yang, H. Duan, Q. Zhang, N. Barrow-Williams, B. C. Turnbull, B. M. Collins, et al. Applying Deep Learning to Airbnb Search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD'19)*, 2019.

[36] A. Halevy, F. Korn, N. F. Noy, C. Olston, N. Polyzotis, S. Roy, and S. E. Whang. Goods: Organizing Google's Datasets. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'16)*, 2016.

[37] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril,

D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, et al. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*, 2018.

[38] M. Herschel, R. Diestelkämper, and H. Ben Lahmar. A Survey on Provenance: What for? What form? What from? *The VLDB Journal*, 26(6), 2017.

[39] D. A. Holland, U. J. Braun, D. Maclean, K.-K. Muniswamy-Reddy, and M. I. Seltzer. Choosing a Data Model and Query Language for Provenance. In *Proceedings of the 2nd International Provenance and Annotation Workshop (IPAW'08)*, 2008.

[40] M. Hutson. Artificial Intelligence Faces Reproducibility Crisis. *Science*, 359(6377), 2018.

[41] M. Interlandi, K. Shah, S. D. Tetali, M. A. Gulzar, S. Yoo, M. Kim, T. Millstein, and T. Condie. Titian: Data Provenance Support in Spark. *PVLDB*, 9(3):216–227, 2015.

[42] M. Jones. How Do We Address The Reproducibility Crisis In Artificial Intelligence? https://bit.ly/2SHEFy8, 2018.

[43] J. Kobielus. How to Solve AI's Reproducibility Crisis. https://bit.ly/2TuPEvw, 2018.

[44] A. Kumar, R. McCann, J. Naughton, and J. M. Patel. Model Selection Management Systems: The Next Frontier of Advanced Analytics. *SIGMOD Record*, 44(4), 2016.

[45] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: Efficient Online Multiprocessor Replay via Speculation and External Determinism. *ACM Sigplan Notices*, 45(3), 2010.

[46] P. Li, X. Rao, J. Blase, Y. Zhang, X. Chu, and C. Zhang. CleanML: A Benchmark for Joint Data Cleaning and Machine Learning [Experiments and Analysis]. In *arXiv pre-print 1904.09483*, 2019.

[47] D. Logothetis, S. De, and K. Yocum. Scalable Lineage Capture for Debugging Disc Analytics. In *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC'13)*, 2013.

[48] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific Workflow Management and the Kepler System. *Concurrency and Computation: Practice and Experience*, 18(10), 2006.

[49] P. Macko and M. Seltzer. Provenance Map Orbiter: Interactive Exploration of Large Provenance Graphs. In *Proceedings of the 3rd USENIX Workshop on the Theory and Practice of Provenance (TaPP'11)*, 2011.

[50] P. Macko and M. Seltzer. A General-Purpose Provenance Library. In *Proceedings of the 4th USENIX Workshop on the Theory and Practice of Provenance (TaPP'12)*, 2012.

[51] M. Maddox, D. Goehring, A. J. Elmore, S. Madden, A. Parameswaran, and A. Deshpande. Decibel: The Relational Dataset Branching System. *PVLDB*, 9(9):624–635, 2016.

[52] R. Mavlyutov, C. Curino, B. Asipov, and P. Cudre-Mauroux. Dependency-Driven Analytics: A Compass for Uncharted Data Oceans. In *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research (CIDR'17)*, 2017.

[53] H. Miao, A. Chavan, and A. Deshpande. ProvDB: Lifecycle Management of Collaborative Analysis Workflows. In *Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics (HILDA'17)*, 2017.

[54] H. Miao, A. Li, L. S. Davis, and A. Deshpande. Towards Unified Data and Lifecycle Management for Deep Learning. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE'17)*, 2017.

[55] K.-K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. L. MacLean, D. W. Margo, M. I. Seltzer, and R. Smogor. Layering in Provenance Systems. In *Proceedings of the USENIX Annual Technical Conference (ATC'09)*, 2009.

[56] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-Aware Storage Systems. In *Proceedings of the USENIX Annual Technical Conference (ATC'06)*, 2006.

[57] L. Murta, V. Braganholo, F. Chirigati, D. Koop, and J. Freire. noWorkflow: Capturing and Analyzing Provenance of Scripts. In *Proceedings of the 5th International Provenance and Annotation Workshop (IPAW'14)*, 2014.

[58] B. K. Olorisade, P. Brereton, and P. Andras. Reproducibility in Machine Learning-Based Studies: An Example of Text Mining. 2017.

[59] H. Park, R. Ikeda, and J. Widom. RAMP: A System for Capturing and Tracing Provenance in MapReduce Workflows. *PVLDB*, 4(12):1351–1354, 2011.

[60] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, and J. Bacon. Practical Whole-system Provenance Capture. In *Proceedings of the 8th ACM Symposium on Cloud Computing (SoCC'17)*, 2017.

[61] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. Pinplay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs. In *Proceedings of the 8th annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'10)*, 2010.

[62] J. F. Pimentel, J. Freire, V. Braganholo, and L. Murta. Tracking and Analyzing the Evolution of Provenance from Scripts. In *Proceedings of the 6th International Provenance and Annotation Workshop (IPAW'16)*, 2016.

[63] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire. noWorkflow: A Tool for Collecting, Analyzing, and Managing Provenance from Python Scripts. *PVLDB*, 10(12):1841–1844, 2017.

[64] H. E. Plesser. Reproducibility vs. Replicability: A Brief History of a Confused Terminology. *Frontiers in Neuroinformatics*, 11, 2018.

[65] F. Psallidas, Y. Zhu, B. Karlas, M. Interlandi, A. Floratou, K. Karanasos, W. Wu, C. Zhang, S. Krishnan, C. Curino, and M. Weimer. Data Science Through the Looking Glass and What We Found There. In *arXiv pre-print 1912.09536*, 2019.

[66] C. Sar and P. Cao. Lineage File System. http://crypto.stanford.edu/cao/lineage.html, 2005.

[67] S. Schelter, J.-H. Böse, J. Kirschnick, T. Klein, and S. Seufert. Automatically Tracking Metadata and Provenance of Machine Learning Experiments. In *Proceedings of the ML Systems Workshop @ NeurIPS'17*, 2017.

[68] R. P. Spillane, R. Sears, C. Yalamanchili, S. Gaikwad, M. Chinni, and E. Zadok. Story Book: An Efficient Extensible Provenance Framework. In *Proceedings of the 1st USENIX Workshop on the Theory and Practice of Provenance (TaPP'09)*, 2009.

[69] V. Sridhar, S. Subramanian, D. Arteaga, S. Sundararaman, D. Roselli, and N. Talagala. Model Governance: Reducing the Anarchy of Production ML. In *Proceedings of the USENIX Annual Technical Conference (ATC'18)*, 2018.

[70] A. Vahdat and T. E. Anderson. Transparent Result Caching. In *Proceedings of the USENIX Annual Technical Conference (ATC'98)*, 1998.

[71] M. Vartak, H. Subramanyam, W.-E. Lee, S. Viswanathan, S. Husnoo, S. Madden, and M. Zaharia. ModelDB: A System for Machine Learning Model Management. In *Proceedings of the 1st Workshop on Human-In-the-Loop Data Analytics (HILDA'16)*, 2016.

[72] W3C. PROV-DM: The PROV Data Model. https://www.w3.org/TR/prov-dm/, 2013.

[73] P. Warden. The Machine Learning Reproducibility Crisis. https://bit.ly/2v0ynQP, 2018.

[74] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, et al. The Taverna Workflow Suite: Designing and Executing Workflows of Web Services on the Desktop, Web or in the Cloud. *Nucleic Acids Research*, 41(W1), 2013.

[75] A. Zhai, H.-Y. Wu, E. Tzeng, D. H. Park, and C. Rosenberg. Learning a Unified Embedding for Visual Search at Pinterest. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD'19)*, 2019.