ProvDB: Lifecycle Management of Collaborative Analysis Workflows

Hui Miao, Amit Chavan, Amol Deshpande
Department of Computer Science, University of Maryland
{hui,amitc,amol}@cs.umd.edu

ABSTRACT
As data-driven methods are becoming pervasive in a wide variety of disciplines, there is an urgent need to develop scalable and sustainable tools to simplify the process of data science, to make it easier for the users to keep track of the analyses being performed and datasets being generated, and to enable the users to understand and analyze the workflows. In this paper, we describe our vision of a unified provenance and metadata management system to support lifecycle management of complex collaborative data science workflows. We argue that the information about analysis processes and data artifacts can, and should, be captured in a semi-passive manner; and we show that querying and analyzing this information can not only simplify bookkeeping and debugging tasks but also enable a rich new set of capabilities like identifying flaws in the process itself. It can also significantly reduce the time spent in fixing post-deployment problems through automated analysis and monitoring. We have implemented a prototype system, ProvDB, on top of git and Neo4j, and we describe its key features and capabilities.

ACM Reference format:
DOI: http://dx.doi.org/10.1145/3077257.3077267

1 INTRODUCTION
Data-driven methods are becoming increasingly common in a variety of communities, which has resulted in a pressing need for sustainable and scalable tools that facilitate the end-to-end data science process (lifecycle) by making it easy to maintain and share time-evolving datasets; to collaboratively clean, integrate, and analyze datasets; to perform introspective analysis to identify errors in the data science pipelines; and to learn from others. This is especially challenging as the collaborative data science lifecycle is often ad hoc, typically featuring highly unstructured datasets, an amalgamation of different tools and techniques, significant back-and-forth among team members, and trial-and-error to identify the right analysis tools, models, and parameters. Although there is much prior and ongoing work on developing systems to perform specific data analysis tasks such as wrangling, training, saving, A/B testing, etc., support for lifecycle management is largely absent in today’s data science platform offerings. This is rapidly becoming a crucial omission since a large and increasing fraction of the overall human attention during the analysis process is being devoted to these issues. In most cases, there is no easy way for the users to capture and reason about ad hoc data science pipelines, many of which are often spread across a collection of analysis scripts. Metadata or provenance information about how datasets were generated, including the user inputs, the steps taken by the user, the scripts used and their versions, and/or values of any crucial parameters, is often lost. Similarly, it is hard to keep track of any dependencies between the artifacts. As most datasets and analysis scripts evolve over time, there is also a need to keep track of their versions over time; using version control systems (VCS) like git can help to some extent, but those don’t provide sufficiently rich introspection capabilities.

Lacking platform support for capturing and analyzing such lifecycle provenance and metadata information, practitioners are required to manually track and act upon it, which is not only tedious, but error-prone. For example, (a) they must manually keep track of which derived datasets need to be updated when a source dataset changes – they often use spreadsheets to list parameter combinations tried out when applying a machine learning model; (b) debugging becomes much harder; e.g., a small change in an analysis script may have significant impact on the final result, but identifying that change may be non-trivial, especially in a collaborative setting; (c) ’repeatability’ can often be very difficult, even for the same practitioner, because of an amalgamation of constantly evolving tools and datasets being used, and a lack of easy-to-use mechanism to keep track of parameter values used in the lifecycle; (d) critical errors may be hidden in the mess of artifacts that cannot be easily identified; e.g., a data scientist may erroneously train on the test dataset due to mistakes while creating the dataset splits.

This paper describes a system, called ProvDB, for unified management of all kinds of metadata about collaborative data science workflows that gets generated during a project lifecycle; this includes (a) version lineages of data, scripts, and results (collectively called artifacts), (b) workflow provenance on derivations among artifact snapshots, (c) important context metadata about artifacts, derivations and the project, (d) data provenance of artifact content which may or may not be structured. Our hypothesis is that by combining information about the lifecycle in one place, and making it easy for practitioners to analyze or query it, we can enable a rich set of functionality that can simplify their lives, make it easier to identify and eliminate errors, and decrease the time to obtain actionable insights. This is hardly a new observation, and there has been much prior work on capturing and analyzing provenance in a variety of communities. However, there is still a lack of practical systems that
treat different kinds of provenance and metadata information in a unified manner, and that can be easily integrated in the lifecycle of a data science project. At the same time, the widespread use of data science has brought to the forefront several important and crucial challenges, such as ethics, transparency, reproducibility, etc.; we posit that fine-granularity provenance is a key to addressing them.

**Challenges & Desiderata:** There are however several crucial challenges and to fully exploit those opportunities. First, it is hard to define a schema for the provenance/metadata information a priori, and different users or different lifecycles may wish to capture and analyze different types of such data. Second, we must be able to ingest the information with minimal involvement from the users and allow them using preferred tools. Third is to come up with a set of query facilities for potential ways to use such data including: (a) explanation queries which help to understand the project, look for origins of certain piece of data, etc., (b) introspection queries that attempt to identify flaws from the lifecycle history (e.g., p-value hacking), (c) continuous monitoring to identify issues of new changes during a lifecycle (e.g., concept drifts where a learned model doesn’t fit new data; changes to input data formats), and many others. Finally, we expect efficiency and optimization issues that will arise as the volume of the captured data increases.

**ProvDB & Outline:** ProvDB is being developed in conjunction with DataHub [5], a dataset-centric platform for enabling collaborative data analytics that supports managing a large number of datasets, their versions over time, and derived data products. Now ProvDB is built on top of git, widely used by practitioners due to its intuitive support for collaboration, and Neo4j, a graph database. As DataHub matures, we plan to integrate ProvDB with it in future.

We begin with discussing closely related work and putting our work in context of that in Sec. 2. We present ProvDB system architecture (Sec. 3.1) and its data model (Sec. 3.2), followed by a discussion of provenance ingestion mechanisms that it supports (Sec. 3.3). We then discuss the types of analyses that ProvDB enables and the web browser-based visualization tool (Sec. 3.4).

## 2 PRIOR WORK

**Provenance Systems:** There has been much work on scientific workflow systems [9] over the years, with some of the prominent ones being Kepler [6], Taverna [17], VisTrails [11], Chimera [8], to name a few. They often center around provenance management for a well-defined workflow, but cannot easily handle fast-changing pipelines, thus typically are not suitable for ad hoc data science projects, as clear established pipelines may not exist except in the final, stable stages. Moreover, they typically require specific computational environments which impose a high overhead on users.

Provenance can be captured at different granularities for computer aided tasks [1, 9], all of which have useful utilities for the users. Workflow provenance is often referred to as coarse-grained and may include: a) prospective information about the workflow definition, b) retrospective information about the workflow execution, c) metadata about steps and datasets in a workflow, and d) input/output lineages among steps [24]. On the other hand, in dataflow systems (e.g., SQL, Pig Latin, Spark), data provenance at record level is studied [1, 14, 18]. Previous efforts, such as Burrito [19], Reprozip [2], noWorkflow [13], Lipstick [18], etc., proposed techniques to ingest and represent workflow and data provenance in specific settings. ProvDB aims to combine the two together with version lineages and provide uniform platform for collaborative data science workflows. It is complementary to, and can utilize prior techniques to capture provenance; our focus is primarily on how to exploit that information and provide richer introspection capabilities.

**Collaborative Data Science Systems:** Many researchers find VCS (e.g., git, svn) and related hosting platforms (e.g., GitHub) much more appropriate for their daily needs. They provide transparent support for versioning and sharing, while imposing no constraints on what types of tools can be used for the data processing itself. Though they keep version lineage among committed artifacts, these systems are typically too ‘low-level’, and have very little query facilities or ingestion capability for capturing higher-level workflows or for keeping track of the operations being performed or any kind of provenance information. Their versioning API is based on a notion of files, and is not capable of allowing users to reason about data within versions and the relationships among versions in a holistic manner. On the other hand, a wide range of analytic packages like SAS, Excel, R, and Matlab, or data science toolkits such as IPython, Scikit, and Pandas, are frequently used for performing analysis itself; however, those lack comprehensive data management or collaboration capabilities. ProvDB can be seen as providing rich introspection and querying capabilities those systems lack.

Sharing similar views, two recent projects aim to improve collaborative data science workflows by reducing the cost of metadata collection and management. LabBook [7], a social data science notebook, uses a queryable property graph to manage metadata captured during collaborative analytics and features a web-based app architecture for analyzing the metadata. However, LabBook does not treat versioning as a first-class construct, and does not focus on developing passive provenance ingestion mechanisms or sophisticated querying abstractions as we do here. Ground [10] is a data context service to manage all the information that informs the use of data. It has a general data model and architecture to import from and export to other systems. However, metadata ingestion and useful high-level query facilities are left to the users.

**Lifecycle Management Systems for Machine Learning:** There has been significant interest in developing systems for handling different aspects of model lifecycle management, such as designing general-purpose training systems like GraphLab, TensorFlow, Parameter Server, etc., accelerating specific modeling tasks (e.g., feature engineering [23], deep learning [22], data science notebook [20]), selecting models [4], and querying models [15]. In contrast, our focus is on the provenance aspect when multiple practitioners collaboratively develop a model, and ProvDB can be used as the provenance management layer for most above systems.

## 3 PROVDB OVERVIEW

### 3.1 System Architecture

ProvDB is a stand-alone client-server system, designed to be used in conjunction with a dataset version control system (DVCS) like git or DataHub (Fig. 1). The DVCS will handle the actual version management tasks, including supporting the standard functionality, i.e., checkout, commit, merge, etc., and the distributed and decentralized management of individual repositories.
We envision a number of local DVCS ‘repositories’, each corresponding to a team of practitioners collaborating closely together. The repository will typically be replicated across a number of machines as different users ‘check out’ the repository contents to work locally. Since we leverage git for keeping these in sync, the repository contents are available as files for the users to operate upon; they can run whichever analysis tools they want on those after checking them out, even distributed ones like Hadoop or Spark.

A repository consists of a set of versions. A version, identified by an ID, is immutable and any update to it conceptually results in a new version with a different version ID (physical data structures are typically not immutable and the underlying DVCS uses various strategies for compact storage [16]). The version-level provenance that captures these processes is maintained as a ‘version graph’, a directed acyclic graph with versions as nodes. Typically, the leaves of the version graph correspond to different live branches that different users may be operating upon at the same time. As we discuss in the next section, ProvDB actually maintains a conceptual ‘workflow graph’ with many other types of nodes and edges.

Broadly, the data maintained across the system can be categorized into: (a) raw data that the users can directly access and analyze including the datasets, analysis scripts, and any derived artifacts such as trained models, and (b) metadata or provenance information transparently maintained by the system. Note that, the split design that we have chosen requires duplication of some information in the DVCS and ProvDB. We believe it is a small price to pay for the benefits of having a standalone provenance management system.

Data Collection Layer is a thin layer on top of the DVCS that is used to capture the provenance and metadata information. This layer needs to support a variety of functionality to make it easy to collect a large amount of metadata and provenance information, with minimal overhead to the user (Sec. 3.3). The ProvDB instance itself is a separate process, and currently uses the Neo4j graph database to store the data; we chose Neo4j because of its support for the flexible property graph data model, and graph query functionality out-of-the-box (Sec. 3.2). The stored data can be accessed either through the Neo4j frontend, or through a visual frontend that we have built that supports a variety of provenance queries (Sec. 3.4).

3.2 Provenance Data Model
To encompass a large variety of situations, our goal was to have a flexible data model that reflects versioning and workflow pipelines, and supports addition of arbitrary metadata or provenance information. As such, we advocate a ‘schema-later’ approach, where a fixed ‘base schema’ (Fig. 2(a)) for capturing information about versions, the different artifacts, and so on, while allowing arbitrary properties to be added to various entities. We store the conceptual model physically as a property graph (Fig. 2(c)), primarily to enable graph traversal queries and visual exploration over the stored information easily (Sec. 3.4). The data model refines the versioning model proposed in our prior work [3], and differs from other similar ones [7, 12, 24] mainly in the explicit modeling of versions.

Conceptual Data Model: We view a data science project as a working directory with a set of artifacts (files), and a development lifecycle as a series derivations (shell commands, edits, programs) performing create/read/update/delete operations in the directory.

More specifically: an artifact is a file, which the user modifies, runs, and talks about with peers. Artifacts can be tagged as belonging to one of three different types: ResultFile, DataFile, ScriptFile, which helps with formulating appropriate queries. A version is a checkpoint of the project; in our case, this refers to a physical commit created via git. ProvDB has explicit versions and implicit versions; the former are created when a user explicitly issues commit command, whereas the latter are created at provenance ingestion time when the user runs commands in the project directory.

Snapshots are checkpointed versions of an artifact and capture its evolution lineage as the parent relationship. The content of a snapshot is modeled as records, to allow fine-grained provenance.

Derivations capture the transformation context to the extent possible. If a derivation is performed by running a program or a script, then the execution history is captured along with any arguments. Derivation edges may also be created when ProvDB notices that one or more artifacts have changed before the transformation (e.g., an edit in an IDE, or a script ran outside the ProvDB context).

Finally, properties are used to encode any additional information about the snapshots or the derivations, as key-value pairs (where values are often time series or JSON documents themselves). Provenance ingestion tools (Sec. 3.3) will generate these properties, which may include any information captured by parsing shell scripts or analysis scripts themselves. Properties can be statistics about the snapshots data as well, so that they can be seamlessly queried. This starts blurring the distinction between data and metadata to some extent; we plan to investigate using a more elaborate data model that more clearly delineates between the two in future.

Physical Property Graph Data Model: We map the conceptual data model (with the exception of Record) into a property graph data model. Nodes of the property graph are of types Version, Artifact, etc., whereas the edges capture the relationships (e.g. parent).

Example 3.1. In Fig. 2(b), a user starts an analysis using a script file script1 and a data file datafile1 by copying them to a repository. She first tries out script1 on datafile1, and a result file result1 with m records is generated. On inspecting result1, she finds a number format issue which she corrects by editing script1 using vim and running script1 again on datafile1. This affects all records in result1. Assuming ProvDB made a commit at the end of each shell interaction, we show the versions, artifacts, snapshots, and derivations. Between the versions, the command is captured as a Derivation, whose properties would be the arguments (i.e. options, parameters). If there is change before a derivation, ProvDB detects it and marks the derivation as missing provenance. In Fig. 2(c), we show the actual physical property graph. Between artifacts and snapshots, (e.g. result1 and result s1), the edge has a composition relationship, while between snapshots, the parent edge is stored across versions.
3.3 Provenance Ingestion

ProvDB captures lifecycle information opportunistically, and features a suite of mechanisms that can capture provenance/metadata for different types of artifacts and derivations. Users can easily configure and add ingestion mechanisms, to change or extend the ingestion capability. Current ProvDB prototype implementation includes: (a) a general-purpose UNIX shell-based ingestion framework, (b) DVCS versioning information importer, (c) user annotation GUI, and (d) a mechanism called *file views*, intended to both simplify workflow and aid in fine-grained provenance capture.

**Shell command-based Ingestion Framework:** Current ProvDB prototype is centered around the UNIX commandline shell (e.g., bash, zsh, etc). A special command called *provdb ingest* that users can prefix to any other command, and that triggers provenance ingestion. Each run of the command results in creation of a new *implicit* version, which allows us to capture the changes at a fine granularity. A collection of *ingestors* is invoked by matching the command against a set of regular expressions, registered a priori along with the ingestors. ProvDB schedules ingestor to run before/during/after execution the user command, and expects the ingestor to return a JSON property graph consisting of a set of key-value pairs denoting properties of the snapshots or derivations.

A default ingestor handles arbitrary commands by parsing them following POSIX standard (IEEE 1003.1-2001) to annotate utility, options, option arguments and operands. For example, `mkdir -p` and `dir` is parsed as utility `mkdir`, option `p` and operand `dir`. Concatenations of commands are decomposed and ingested separately, while a command with pipes is treated as a single command. If an external tool has been used to make any edits (e.g., a text editor), an implicit version is created next time *provdb ingest* is run, and the derivation information is recorded as missing. ProvDB also supports several specialized ingestion plugins and configurations to cover important data science workflows. In particular, it has an ingestor capable of ingesting provenance information from runs of the *caffe* deep learning framework; it not only ingests the learning hyperparameters from the configuration file, but also the accuracy and loss scores by iteration from the result logging file. We are currently working on ingestors for scripts written in popular data science tools such as *scikit-learn* and *pandas* as in [13, 15].

**User Annotations:** Context metadata, cognitive annotations and communications are important for collaborative data science [7, 10, 20]. ProvDB GUI allows users to organize, add, and annotate properties, along with other query facilities. Users can annotate project properties, such as usage descriptions for collaborations on artifacts, or notes to explain rationale for a particular derivation. A user can also annotate a property as parameter and add range/step to its domains, which turns a derivation into a template and enables batch run of an experiment. For example, a grid search of a template derivation on a start snapshot can be configured directly in the UI. Maintaining such user annotations (and file views discussed next) as the datasets evolve is a complicated issue in itself [21].

**File Views:** ProvDB provides a functionality called *file views* to assist dataset transformations and to ingest provenance among data files. Analogous to views in relational databases, a file view defines a virtual file as a transformation over an existing file. A file view can be defined either: (a) as a script or a sequence of commands (e.g., `sort | uniq -c`, which is equivalent to an aggregate count), or (b) as an SQL query where the input files are treated as tables. For instance, the following query counts the rows per label that a classifier predicts wrongly comparing with ground truth. **provdb fileview -c -n=’results.csv’ -q=’**

```bash
select t.c2 as label, count(*) as err_cnt
from (testfile.csv) as t, (predfile.csv) as r
where t.c0 = r.c0 and t.c2 != r.c2 group by t.c2'
```

The SQL feature is implemented by loading the input files into an in-memory *sqlite* database and executing the query against it. Instead of creating a view, the same syntax can be used for creating a new file instead, saving a user from coding similar functionality.

File views serves as an example of a functionality that makes the ad hoc process of data science more structured. Aside from making it easier to track dependencies, SQL-based file views also enable capturing record-level provenance by drawing upon techniques developed over the years for data provenance in databases [1].

**Discussion:** Currently ProvDB can be used in a command-line environment. In future work, we plan to investigate ingestion within other development environments such as different IDEs and important apps [19]. We also plan to incorporate support for ingesting log files generated in many environments today, and through continuous monitoring of the artifacts in the working directory.

### 3.4 Query and Analysis Facilities

The major data management research challenges in building a system like ProvDB revolve around querying, analyzing, and extracting insights from the rich provenance information collected using the mechanisms described so far. In addition to *explanation queries* which look for origins of a piece of data and *explorative lifecycle*
queries on the property graph, ProvDB enables asking deeper, introspective queries about the data science processes and pipelines, and formalizing those is a major challenge in itself. ProvDB can also naturally support monitoring queries, which can be used to automatically detect problems during deployment. We hope that building the basic infrastructure to collect and expose the information will allow other researchers and data scientists to start formulating such questions more easily. Developing a higher-level query language also remains a major challenge; although we proposed an initial design of a query language in our prior work [3], it does not support querying over workflow derivations or analysis artifacts.

**Queries over Version/Workflow Graph and Properties:** In a collaborative workflow, provenance queries to identify what revision and which author last modified a line in an artifact are common (e.g., `git blame`). ProvDB allows such queries on the version graph and supports rich versions queries [3]. Moreover, queries can be asked at various levels (version, artifact, snapshot, record) on both the version graph and the workflow graph, and using properties associated with the different entities (e.g., details of what parameters have been used, temporal orders of commands, etc). In fact, all the information exposed in the property graph can be directly queried using the Neo4j Cypher query language.

The capability of the queries using properties are primarily limited by the amount of information that can be automatically ingested. Using the current ingestors (Sec. 3.3), such as a program analysis ingester for scikit-learn which extracts the scikit-learn APIs used in a script, and a hyper-parameter and result-table ingester for caffe for deep learning (the hyper-parameter ingester extracts experiment parameters from caffe commands and arguments, while the results-table ingester extracts errors and accuracies from training logs), meaningful queries can be asked, e.g. which scikit-learn script contains a specific sequence of commands; what is the learning accuracy curve of a deep learning model; enumerate all parameter combinations that have been tried out for a given learning task, etc.

**Shallow vs Deep “Diff” Queries:** “Diff” is a first-class operator in ProvDB, and can be used for finding differences at various different levels. Specifically, given a pair of nodes (corresponding to two snapshots) in the property graph, a shallow diff operation, by default, focuses on the ingested properties of the two snapshots, which are likely to contain the crucial differences in most cases. It attempts to “join” the two sets of properties as best as it can, and highlights the differences; in case of time-series properties, it also allows users to generate plots so they can more easily understand the differences. For example, for two `result table artifacts` that may represent the outputs of two different runs of the same script (e.g., model training logs), a line-by-line diff may be useless because of irrelevant and minor numerical differences; however, by plotting the two sets of results against each other, a user can more quickly spot important trends (e.g., that a specific value of parameter leads to quicker convergence). The shallow diff operation also allows differing contents of two files line-by-line if so desired.

A deep diff compares the ancestors of the two target snapshots by tracing back their derivations to the common ancestor. It aligns the snapshots along the two paths, and shows the differences between each pair of aligned snapshots. For example, in a prediction task, a user may have tried out different models or configurations to improve the test accuracy; in ProvDB, she can start from two result files, and ask a deep diff query to compare how they are derived.

**Record Provenance Queries:** Although the ProvDB data model supports storing fine-grained record-level provenance information, it currently does not have an ingestor that generates such data; we are working on adding several such ingestors, including ones for SQL-based file views or transformations, and for common data cleaning or similar operations where record-level provenance can be easily inferred. Given such information, record-level provenance queries are conceptually straightforward. However, the main challenge is expected to be the large volume of provenance information as well as efficient query execution. We plan to investigate approximate (lossy) provenance storage mechanisms to address these challenges. The utility of these queries may also be limited because it is difficult to collect fine-grained provenance for many black-box operations (e.g., machine learning models). Developing techniques to do so remains a rich area for further work.

**Reasoning about Pipelines:** Similar to a workflow management system, we define a pipeline to be a sequence of derivation edges. A pipeline can be annotated by the user by browsing the workflow graph and marking the start and the end edges of the pipeline. Pipelines can also be inferred automatically by the system (e.g., via pattern mining techniques). ProvDB UI allows a user to browse and reuse pipelines present in the system; in future, we also plan to add support for re-invoking an old pipeline on an old artifact to verify the results, or invoking a pipeline on a different snapshot with different parameters, or schedule a cron job. Being able to reason about pipelines has the potential to hugely simplify the lives of data scientists, by allowing them to learn from others and also helping them avoid mistakes (e.g., omission of a crucial intermediate step).

**Continuous Monitoring or Anomaly Detection:** On ingested properties of artifacts and derivations, ProvDB provides a monitoring and alerting subsystem. We envision two main introspection scenarios for this functionality: (a) it can be used to detect any major changes to the properties of an evolving dataset – e.g., a large change in the distribution of values in a dataset may be cause for taking remedial action. (b) In most applications, there is usually a need to “deploy” an analysis script or a trained model against live incoming data; it is important to keep track of how well the model or the script is behaving and catch any problems as soon as possible (e.g., changing input data properties; higher error rates than expected). Currently even if systems like Spark Streaming or Apache Storm can be used to execute a script against new data in a streaming fashion, there is no built-in support for the introspection tasks. Newer systems like Google TensorFlow Serving also facilitate the deployment process, but do not support introspection. Such introspection can be seen as continuous queries against streaming provenance information. Currently, ProvDB supports simple alert queries that can monitor a property on an evolving artifact through the ProvDB web GUI; in future iterations of ProvDB, we plan to support more complex temporal queries (that can monitor properties across snapshots) and we plan to support executing those continuously as new versions (implicit or explicit) are checked in.

**Discussion:** One major challenge that we do not discuss further has to do with efficiency; in real deployments with large collaboration teams, a large amount of provenance information may be
collected that will likely overwhelm a single, centralized instance (especially if record-level provenance information is collected). New techniques for compressing the information, and maintaining the information in a distributed manner would need to be developed.

Example 3.2. We show a concrete ProvDB query scenario on a deep learning repository, in which 41 neural networks are created for a face classification task. The models are enumerated by mimicking modeling practices by varying networks and parameters. In Fig 3(a), the user filters modeling artifacts and selects two models (model-0 and 9) using left pane, then issues an introspection query asking about their differences. Using the GUI, the user diffs their ingested provenance properties from caffe logging files. The right query result pane highlights the differences in the ingested properties. The caffe ingester properties are numerical time series; using the provided charting tool, the user plots the training loss and accuracy against the iteration number. From the results, we can see that model-9 does not train well in the beginning, but ends up with similar accuracy. To understand why, a deep diff between the two can be issued in the GUI and complex Cypher queries can be used as well. In Fig. 3(b), the query finds previous derivations and shared snapshots, which are training config files; more introspection can be done such as finding changed hyperparameters.

4 CONCLUSION

In this paper, we presented our vision for a system to simplify lifecycle management of ad hoc, collaborative analysis workflows that are becoming prevalent in most application domains today. We argued that a large amount of provenance and metadata information can be captured passively, and analyzing it in novel ways can immensely simplify the day-to-day processes undertaken by data analysts. We have built an initial prototype using git and Neo4j, which provides a variety of provenance ingestion mechanisms and the ability to query, analyze, and monitor the captured provenance information. Our initial experience with using this prototype for a deep learning workflow (for a computer vision task) shows that even with limited functionality, it can simplify the bookkeeping tasks and make it easy to compare the effects of different hyperparameters and neural network structures. However, many interesting and hard systems and conceptual challenges remain to be addressed in capturing and exploiting such information to its fullest extent.

REFERENCES