ProvDB: Provenance-enabled Lifecycle Management of Collaborative Data Analysis Workflows

Hui Miao, Amol Deshpande
University of Maryland, College Park, MD, USA
{hui, amol}@cs.umd.edu

Abstract

Collaborative data science activities are becoming pervasive in a variety of communities, and are often conducted in teams, with people of different expertise performing back-and-forth modeling and analysis on time-evolving datasets. Current data science systems mainly focus on specific steps in the process such as training machine learning models, scaling to large data volumes, or serving the data or the models, while the issues of end-to-end data science lifecycle management are largely ignored. Such issues include, for example, tracking provenance and derivation history of models, identifying data processing pipelines and keeping track of their evolution, analyzing unexpected behaviors and monitoring the project health, and providing the ability to reason about specific analysis results. In this article, we present an overview of a unified provenance and metadata management system, called PROVDB, that we have been building to support lifecycle management of complex collaborative data science workflows. PROVDB captures a large amount of fine-grained information about the analysis processes and versioned data artifacts in a semi-passive manner using a flexible and extensible ingestion mechanism; provides novel querying and analysis capabilities for simplifying bookkeeping and debugging tasks for data analysts; and enables a rich new set of capabilities like identifying flaws in the data science process.

1 Introduction

Over the last two decades, we have seen a tremendous increase in collaborative data science activities, where teams of data engineers, data scientists, and domain experts work together to analyze and extract value from increasingly large volumes of data. This has led to much work in industry and academia on developing a wide range of tools and techniques, including big data platforms, data preparation and wrangling tools, new and sophisticated machine learning techniques, and new environments like Jupyter Notebooks targeted at data scientists. However, the widespread use of data science has also exposed the shortcomings of the overall ecosystem, and has brought to the forefront a number of critical concerns about how data is collected and processed, and how insights or results are obtained from those. There are numerous examples of biased and unfair decisions made by widely used data science pipelines, that are often traced back to flaws in the training data, or mistakes or assumptions made somewhere in the process of constructing those pipelines. Data analysis processes are relatively easy to "hack" today, increasingly resulting in a general lack of trust in published results ("fake news") and many incidents of unreproducible and/or retracted results.

Addressing these concerns requires a holistic approach to conducting, auditing, and continuously monitoring collaborative data science projects. This however is very challenging, because of the fundamentally ad hoc nature of collaborative data science. It typically features highly unstructured and noisy datasets that need to be cleaned and wrangled; amalgamation of different platforms, tools, and techniques; significant back-and-forth among the members of a team; and trial-and-error to identify the right analysis tools, algorithms, machine learning models, and parameters. The data science pipelines are often spread across a collection of analysis scripts, possibly across different systems. Metadata or provenance information about how datasets were generated, including the programs or scripts used for generating them and/or values of any crucial parameters, is often lost. 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 systematic platforms to support collaborative data science, we see users manually track and act upon such information. For example, users often manually keep track of which derived datasets need to be updated when a source dataset changes; they use spreadsheets to list which parameter combinations have been tried out during the development of a machine learning model; and so on. This is however tedious and error-prone, requires significant discipline from the users, and typically lacks sufficient coverage.

In this article, we present an overview of our PROVDB system, for unified management of all kinds of metadata about collaborative data science workflows that gets generated during the analysis processes. PROVDB is being developed under the umbrella of the **DataHub** project [13], where our overarching goal is to understand and investigate the major pain points for data engineers and scientists today, and develop tools and techniques for managing a large number of datasets, their versions over time, and derived data products and for supporting the end-to-end data science lifecycle. PROVDB is designed to be a standalone system, with complementary functionality to dataset versioning tools developed in [14, 15, 17, 28, 33]. Specifically, PROVDB manages and enables querying information about: **(a)** version lineages of data, scripts, and results (collectively called *artifacts*), **(b)** data provenance among artifacts which may or may not be structured, and **(c)** workflow metadata on derivations and dependencies among artifact snapshots, and **(d)** other context metadata that may be collected (e.g., summary logs). By making it easy to mine and query this information in a unified fashion, PROVDB enables a rich set of functionality to simplify the lives of data scientists, to make it easier to identify and eliminate errors, and to decrease the time to obtain actionable insights.

To accommodate the wide variety of expected use cases, PROVDB adopts a "schema-later" approach, where a small base schema is fixed, but users can add arbitrary semistructured information (in JSON) for recording additional metadata. The specific data model we use generalizes the standard W3C PROV data model, extended to allow flexibly capturing a variety of different types of information including versioning and provenance information, parameters used during experiments or modeling, statistics gathered to make decision, analysis scripts, and notes or tags. We map this logical data model to a *property graph data model*, and use the Neo4j graph database to store the information.

PROVDB features a suite of extensible *provenance ingestion* mechanisms. The currently supported ingestors are targeted at users who conduct the data analysis using either `UNIX Shell` or `Jupyter Notebooks`. In the former case, user-run `shell` commands to manipulate files or datasets are intercepted, and the before- and after-state of the artifacts is analyzed to generate rich metadata. Several such ingestors are already supported and new ingestors can be easily registered for specific commands. For example, such an ingestion program is used to analyze log files generated by Caffe (a deep learning framework) and to generate metadata about accuracy and loss metrics (for learned models) in a fine-grained manner. PROVDB also features APIs to make it easy to add provenance or context information from other sources.

Finally, PROVDB features many different mechanisms for querying and analyzing the captured data. In addition to being able to use the querying features of `Neo4j` (including the Cypher and Gremlin query languages), PROVDB also features two novel graph query operators, *graph segmentation* and *graph summarization*, that are geared towards the unique characteristics of provenance information.

2 ProvDB Architecture

PROVDB is a stand-alone system, designed to be used in conjunction with a dataset version control system (DVCS) like `git` or `DataHub` [13, 14, 17, 18] (Figure 1). The DVCS will handle the actual version management tasks, and the distributed and decentralized management of individual *repositories*. A repository consists of a set of *versions*, each of which is comprised of a set *artifacts* (scripts, datasets, derived results, etc.). A version, identified by an ID, is <u>immutable</u> and any update to a version conceptually results in a new version with a different version ID (physical data structures are typically not immutable and the underlying DVCS may use various strategies for compact storage [14]). New versions can also be created through the application of transformation programs to one or more existing versions. We assume the DVCS supports standard version control operations including CHECKOUT a version, COMMIT a new version, create a BRANCH, MERGE versions, etc.

Most of the PROVDB modules are agnostic to the DVCS being used, as long as appropriate pipelines to transfer versioning metadata and context data are set up. The current implementation is based on `git`, in which case, the repository contents are available as files for the users to operate upon; the users can run whichever analysis tools they want on those after checking them out, including distributed toolkits like Hadoop or Spark.

Broadly speaking, the data maintained across the system can be categorized into:

Figure 1: Architecture Overview of PROVDB

(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, and used for answering queries over the versioning or provenance information.

Fine-grained record-level provenance information may or may not be directly accessible to the users depending on the ingest mechanism used. Note that, the split design that we have chosen to pursue requires duplication of some information in the DVCS and PROVDB. We believe this is a small price to pay for the benefits of having a standalone provenance management system.

PROVDB **Ingestion module** is a thin layer on top of the DVCS (in our case, `git`) 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. 4). The PROVDB instance itself is a separate process, and currently uses the Neo4j graph database for **provenance model storage**; we chose Neo4j because of its support for the flexible property graph data model, and graph querying functionality out-of-the-box (Sec. 3). The data stored inside PROVDB can be queried using Cypher through the Neo4j frontend; PROVDB **query execution engine** also supports two novel types of *segmentation* and *summarization* queries that we breifly describe in Section 5. Richer queries can be supported on top of these low-level querying constructs; PROVDB currently supports a simple form of continuous monitoring query, and also has a visual frontend to support a variety of provenance queries (Section 5).
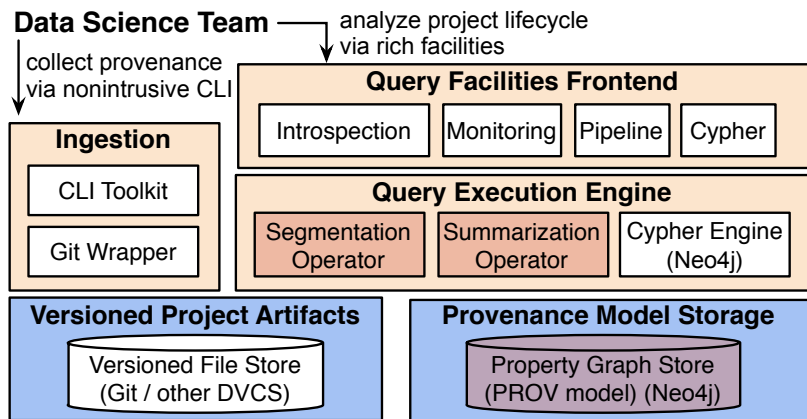
	Alice commit (v1)			Alice commit (v2)			Bob commit (v3)	
Activity	**Entity**	**Property**	**Activity**	**Entity**	**Property**	**Activity**	**Entity**	**Property**
copy	dataset	url: http://.						
create	model	ref: vgg16	update	model	ann: AVG			
create	solver	iter: 20000				update	solver	lr: 0.01
train		opt: -gpu	train		opt: -gpu	train		opt: -gpu
	logs	acc: 0.7		logs	acc: 0.5		logs	acc: 0.75
	weight	exp: v1		weight	exp: v2		weight	exp: v3

(a) An Example for a Data Science Project Lifecycle & Associated Provenance

(b) Illustration of the W3C PROV Data Model

(c) Provenance Graph for the Lifecycle Example (Some edges and properties are not shown due to space constraints)
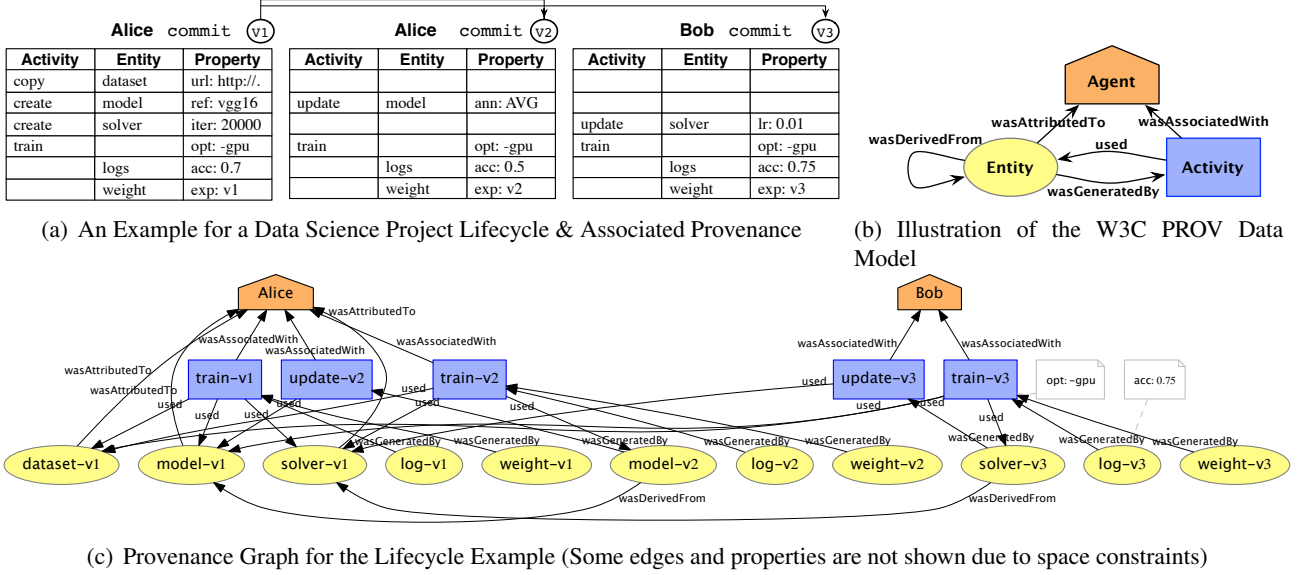
Figure 2: Illustration of Provenance Data Model and Query Operators in Data Science Lifecycles

3 Provenance Data Model and Provenance Graphs

The ingested provenance over the lifecycle of a data science project naturally forms a directed acyclic graph[1], combining heterogeneous information including a version graph representing the artifact changes, a workflow graph reflecting the derivations of those artifact versions, and a conceptual model graph showing the involvement of problem solving methods in the project [21, 30]. To represent the provenance graph and keep our discussion general to other provenance systems, after originally using a custom model, we switched to using the W3C PROV data model [35], which is a standard interchange model for different provenance systems. We use the core set of PROV data model shown in Fig. 2(b). There are three types of vertices ($\mathbb{V}$) in the provenance graph in our context:

a) <u>Entities</u> ($\mathcal{E}$) are the project artifacts (e.g., files, datasets, scripts) which the users work on and talk about;

b) <u>Activities</u> ($\mathcal{A}$) are the system or user actions (e.g., `train`, `git commit`, `cron` jobs) which act upon or with entities over a period of time, $[t_i, t_j)$;

c) <u>Agents</u> ($\mathcal{U}$) are the parties who are responsible for some activity (e.g., a team member, a system component).

Among the vertices, there are five types of directed edges ($\mathbb{E}$):

i) An activity started at time t_i often uses some entities ('<u>used</u>', $U \subseteq \mathcal{A} \times \mathcal{E}$);

ii) then some entities were generated by the same activity at time t_j ($t_j \geq t_i$) ('<u>wasGeneratedBy</u>', $G \subseteq \mathcal{E} \times \mathcal{A}$);

iii) An activity is associated with some agent during its period of execution ('<u>wasAssociatedWith</u>', $S \subseteq \mathcal{A} \times \mathcal{U}$);

iv) Some entity's presence can be attributed to some agent ('<u>wasAttributedTo</u>', $A \subseteq \mathcal{E} \times \mathcal{U}$); and

v) An entity was derived from another entity ('<u>wasDerivedFrom</u>', $D \subseteq \mathcal{E} \times \mathcal{E}$), such as versions of the same artifact (e.g., different model versions in v_1 and v_2 in Fig. 2(a)).

---

[1]We use versioning to avoid cyclic self-derivations of the same entity and overwritten entity generations by some activity.

4

In the provenance graph, both vertices and edges have a label to encode their vertex type in $\{\mathcal{E}, \mathcal{A}, \mathcal{U}\}$ or edge type in $\{U, G, S, A, D\}$. All other ingested <u>provenance records</u> are modeled as properties, that are ingested by a set of configured project ingestors during activity executions and represented as key-value pairs.

PROV standard defines various serializations of the concept model (e.g., RDF, XML, JSON) [37]. In our system, we use a physical property graph data model to store it, as it is more natural for the users to think of the artifacts as nodes when writing queries using Cypher or Gremlin. It is also more compact than RDF graph for the large amount of provenance records, which are treated as literal nodes.

Definition 1 (Provenance Graph): Provenance in a data science project is represented as a directed acyclic graph, $\mathcal{G}(\mathbb{V}, \mathbb{E}, \lambda_v, \lambda_e, \sigma, \omega)$, where vertices have three types, $\mathbb{V} = \mathcal{E} \cup \mathcal{A} \cup \mathcal{U}$, and edges have five types, $\mathbb{E} = U \cup G \cup S \cup A \cup D$. Label functions, $\lambda_v: \mathbb{V} \mapsto \{\mathcal{E}, \mathcal{A}, \mathcal{U}\}$, and $\lambda_e: \mathbb{E} \mapsto \{U, G, S, A, D\}$ are total functions associating each vertex and each edge to its type. In a project, we refer to the set of property types as $\mathcal{P}$ and their values as O, then vertex and edge properties, $\sigma: \mathbb{V} \times \mathcal{P} \mapsto O$ and $\omega: \mathbb{E} \times \mathcal{P} \mapsto O$, are partial functions from vertex/edge and property type to some value.

Example 1: In Fig. 2(a), Alice and Bob work together on a classification task to predict face IDs given an image. Alice starts the project and creates a neural network by modifying a popular model. She downloads the dataset and edits the model definitions and solver hyperparameters, then invokes the training program with specific command options. After training the first model, she examines the accuracy in the log file, annotates the weight files, then commits a version using `git` via ProvDB command-line interface (CLI, Fig. 1). As the accuracy of the first model is not ideal, she changes the network by editing the model definition, trains it again and derives new log files and weight parameters. However the accuracy drops, and she turns to Bob for help. Bob examines what she did, trains a new model following best practices by editing the learning rate in solver configuration in version v_1.

Behind the scenes, ProvDB tracks users' activities, ingests provenance, and manages versioned artifacts (e.g., datasets, models, solvers). In the Fig. 2(a) tables, we show ingested information in detail: *a)* history of user activities (e.g., the first *train* command uses model v_1 and solver v_1 and generates logs v_1 and weights v_1), *b)* versions and changes of entities (e.g., weights v_1, v_2 and v_3) and derivations among those entities (e.g., model v_2 is derived from model v_1), and *c)* provenance records as associated properties to activities and entities (e.g., dataset is copied from some url, Alice changes a pool layer type to AVG in v_2, accuracy in logs v_3 is 0.75).

Example 2: In Fig. 2(c), we show the corresponding provenance graph of the project lifecycle illustrated in Example 1. Names of the vertices (e.g., 'model-v1', 'train-v3', 'Alice') are made by using their representative properties and suffixed using the version ids to distinguish different snapshots. Activity vertices are ordered from left to right w.r.t. the temporal order of their executions. We label the edges using their types and show a subset of the edges in Fig. 2(a) to illustrate usages of five relationship types. Note there are many snapshots of the same artifact in different versions, and between the versions, we maintain derivation edges 'wasDerivedFrom' (*D*) for efficient versioning storage. The property records are shown as white boxes but not treated as vertices in the property graph.

4 Capturing and Ingesting Provenance

The biggest challenge for a system like ProvDB is capturing the requisite provenance and context information. Transparent instrumentation approaches, where the information is captured with minimal involvement from the users are most likely to be used in practice, but are difficult to develop (since they must cover different user environments) and may generate large amounts of frivolous information (since they cannot be targetted). On the other hand, approaches that require explicit instrumentation or hints from the users can capture more useful information, but require significant discipline from the users and rarely work in the long run. We use a hybrid

approach in ᴘʀᴏᴠDB, where we transparently instrument some of the most common data science environments, but allow users to explcitly instrument and send additional information to the ᴘʀᴏᴠDB server. Here we briefly enumerate the ingestion mechanisms that ᴘʀᴏᴠDB currently supports, which include a general-purpose UNIX shell-based ingestion framework, ingestion of DVCS versioning information, and a mechanism called **file views** which is intended to both simplify workflow and aid in fine-grained provenance capture.

Shell command-based Ingestion Framework: The provenance ingestion framework is centered around the UNIX commandline shell (e.g., `bash, zsh`, etc). We provide a special command called `provdb` that users can prefix to any other command, and that triggers provenance ingestion (shell extensions can be leveraged to do this automatically for every command). Each run of the command results in creation of a new *implicit* version, which allows us to capture the changes at a fine granularity. In other words, we `commit` a new version before and after running every such command. These implicit versions are kept separate from the explicit versions created by a user through use of `git commit`, and are not visible to the users. A collection of *ingestors* is invoked by matching the command that was run, against a set of regular expressions registered a priori along with the ingestors. ᴘʀᴏᴠDB 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. Note that, it is vital that we take a snapshot before running the command in order to properly handle modifications made using external tools (e.g., text editors) or commands not prefixed with `provdb`. That way we can capture a modification even if we don't know the specific operation(s) that made that modification. An ingestor can also provide *record-level provenance* information, if it is able to generate such information.

A default ingestor handles abitrary commands by parsing them following POSIX standard (IEEE 1003.1-2001) to annotate utility, options, option arguments and operands. For example, `mkdir -p 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` is run, and the derivation information is recorded as missing.

Caffe (Deep Learning Framework) Ingestor: ᴘʀᴏᴠDB 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. If the command is recognized as a `caffe` command, then the ingestor parses the configuration file (passed as an argument to the command) to extract and record the *learning hyperparameters*. The ingestor also extracts accuracy and loss scores on an iteration-by-iteration basis from the result logging file. All of these are attached as properties to the coarse-grained derivation edge that is recorded in the provenance graph.

File Views: ᴘʀᴏᴠDB 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 view), 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='
  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 can help make 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 provenance in databases.

User Annotations: Apart from plugin framework, PROVDB GUI allows users to organize, add, and annotate properties, along with other query facilities. The user 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 above) as the datasets evolve is a complicated issue in itself [25].

## 5 Storing, Querying and Analyzing Provenance Graphs

Provenance graphs represent a unique and new usage scenario for graph databases, that necessitates development of new techniques both for storing and for querying them. In particular, the following characteristics of the provenance graph need to be considered:

- **Versioned Artifacts**: Each entity is a point-in-time snapshot of some artifact in the project, e.g., the query 'accuracy of model-v1' discusses a particular *snapshot* of the model artifact, while 'what are the common updates for *solver* before `train`' refer to the *artifact* but not an individual snapshot. This impacts both storage and declarative query facilities. First, given the potentially large *properties* associated with the nodes in the graph (e.g., scripts, notebooks, log files), the standard model of storing each node independently of the others does not work because of the verbosity and duplication in the properties of the data [29]. Second, any high-level query languages must support versioning as a first-class construct [18].

- **Evolving Workflows**: Data science lifecycle is exploratory and collaborative in nature, so *there is no static workflow skeleton, and no clear boundaries for individual runs* in contrast with workflow systems [20]; e.g., the modeling methods may change (e.g., from SVM to neural networks), the data processing steps may vary (e.g., split, transform or merge data files), and the user-committed versions may be mixed with code changes, error fixes, thus may not serve as query boundaries. Thus, we cannot assume the existence of a workflow skeleton and we need to allow flexible boundary conditions.

- **Partial Knowledge in Collaboration**: Each team member may work on and be familiar with a subset of artifacts and activities, and may use different tools or approaches, e.g., in Example 1, Alice and Bob use different ways to improve accuracy. When querying retrospective provenance of the snapshots or understanding activity process over team behaviors, the user may only have partial knowledge at query time, thus may find it difficult to compose the right query. Hence, it is important to support queries with partial information reflecting users' understanding and induce correct result.

- **Verboseness for Usage**: In practice, the provenance graph would be very verbose for humans to use and in large volume for the system to manage. Depending on the granularity, storing the graphs could take dozens of GBs within several minutes [9]. With similar goals to recent research efforts [2, 22, 34, 36], our system aims to let the users understand the essence of provenance at their preference level by transforming the provenance graph. *R4: The query facility should be scalable to large graph and process queries efficiently.*

In addition, most of the provenance query types of interest involve paths [1], and require returning paths instead of answering yes/no queries like reachability [22]. Writing queries to utilize the lifecycle provenance is beyond the capabilities of the pattern matching query (BPM) and regular path query (RPQ) support in popular graph databases [6, 7, 42]. For example, answering 'how is today's result file generated from today's data file' requires a segment of the provenance graph that includes not only the mentioned files but also others that are not
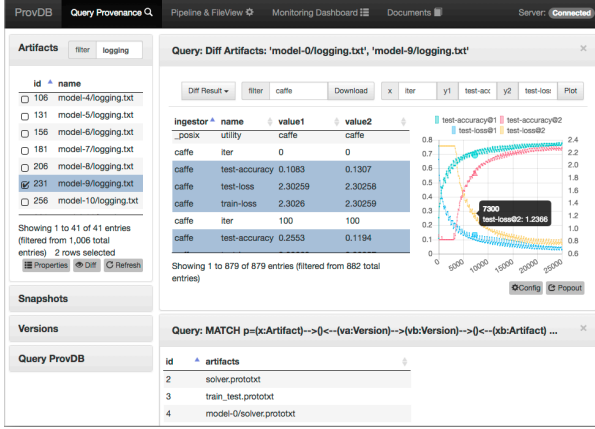
(a) Diff Artifacts (Result logging files for two deep neural networks)
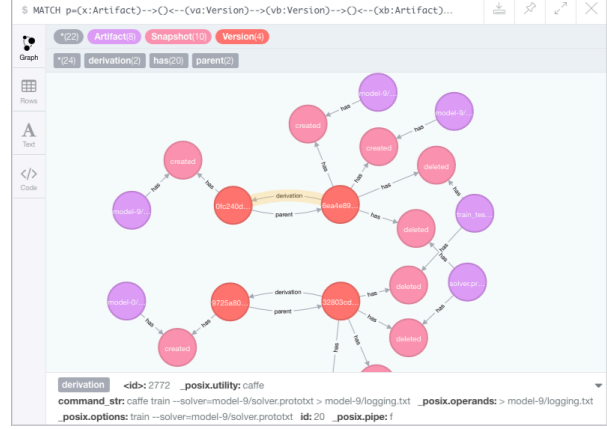
(b) Cypher Query to Find Related Changes via Derivations

Figure 3: Illustration of ProvDB interfaces

on the lineage paths and the users may not know at all (e.g., 'a configuration file'); answering 'how do the team members typically generate the result file from the data file?' requires summarizing several query results of the above query while keeping the result meaningful from provenance perspective.

This lack of proper query facilities in modern graph databases not only limits the value of lifecycle provenance systems for data science, but also of other provenance systems. The specialized query types of interest in the provenance domain [1, 22] had often led provenance systems to implement specialized storage systems [38] and query interfaces [4, 16] on their own [20]. Recent works in the provenance community propose various task-specific graph transformations, which are essentially different template queries from the graph querying perspective; these include grouping vertices together to handle publishing policies [34], aggregating similar vertices to understand commonalities and outliers [36], segmenting a provenance graph for feature extractions in cybersecurity [2].

To handle the wide variety of usage scenarios, ProvDB currently supports several different mechanisms to interact with the provenance information, that we briefly discuss next.

5.1 Interactive Queries using Cypher

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 at various levels (version, artifact, snapshot, record) and also allows querying the 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, which supports graph traversal queries and aggregation queries.

The latter types of queries are primarily limited by the amount of context and properties that can be automatically ingested. Availability of this information allows users to ask more meaningful queries like: *what scikit-learn script files contain a specific sequence of commands; what is the learning accuracy curve of a caffe model artifact; enumerate all different parameter combinations that have been tried out for a given learning task*, and so on.

Many such queries naturally result in one or more time series of values (e.g., properties of an artifact over time as it evolves, results of "diff" queries discussed below); ProvDB supports a uniform visual interface for plotting such time series data, and comparing two different time series (see below for an example).

Illustrative Example: Figure 3 shows the ProvDB Web GUI using a `caffe` deep learning project. In this

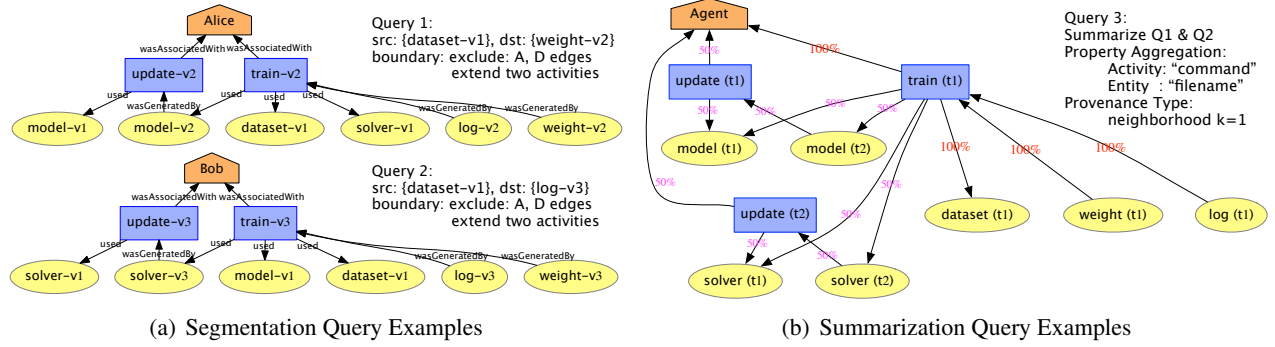(a) Segmentation Query Examples   (b) Summarization Query Examples

Figure 4: Examples of Segmentation and Summarization Queries

project, 41 deep neural networks are created for a face classification task. The user tries out models by editing and training models. In Fig 3(a), an introspection query asks how different are two trained models (*model-0* and *9*). Using the GUI, the user filters artifacts, and diffs their result logging files. In the right side query result pane, the ingested properties are diffed. The `caffe` ingestor 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 by finding changed hyperparameters.

5.2   Graph Segmentation and Summarization Queries

As discussed above, the extant graph query languages are not rich enough to support common provenance analysis queries. One of our goals with ongoing work is to initiate a more systematic study of abstract graph operators that modern graph databases need to support to be a viable option for storing provenance graphs. By observing the characteristics of the provenance graph in analytics lifecycle and identifying the requirements for the query facilities, we propose two new graph operators as a starting point:

Segmentation: A very important provenance query type of interest is querying ancestors and descendants of entities [20, 22]. In our context, the users introspect the lifecycle and identify issues by analyzing dependencies among snapshots. Lack of a workflow skeleton and clear boundaries makes the provenance query more difficult. Moreover the user may not be able to specify all interested entities in a query due to partial knowledge. We propose a segmentation operator that takes sets of source and destination entities, and induces other important unknown entities satisfying a set of specified boundary criteria.

Example 3: In Fig. 4(a), we show two examples of segmentation query. In Query 1 (Q_1), Bob was interested in what Alice did in version v_2. He did not know the details of activities and the entities Alice touched, instead he set {*dataset*}, {*weight*} as querying entities to see how the *weight* in Alice's version v_2 was connected to the *dataset*. He filtered out uninterested edge types (.e.g, *A*, *D*) and excluded actions in earlier commits (e.g., v_1) by setting the boundaries as two activities away from those querying entities. The system found connections among the querying entities, and included vertices within the boundaries. After interpreting the result, Bob knew Alice updated the model definitions in *model*. On the other hand, Alice would understand how Bob improved the accuracy and learn from him. In Query 2 (Q_2), instead of learned *weight*, accuracy property associated *log* entity is used as querying entity along with *dataset*. The result showed Bob only updated solver file.

Summarization: In workflow systems, querying the workflow skeleton (aka prospective provenance) is an important use case [12] and one of the *provenance challenges* [1]. In our context, even though a static workflow skeleton is not present, summarizing a skeleton of similar processing pipelines, showing commonalities and

9

identifying abnormal behaviors are very useful query capabilities. However, general graph summarization techniques [26] are not applicable to provenance graphs due to the subtle provenance meanings and constraints of the data model [34–36]. Hence, we propose a summarization operator with multi-resolution capabilities for provenance graphs. It operates over query results of segmentation and allows tuning the summary by ignoring vertex details and characterizing local structures, and ensures provenance meaning through path constraints.

Example 4: In Fig. 4(b), an outsider to the team (e.g., auditor, new team member) wanted to see the activity overview in the project. Segmentation queries (e.g., Q_1, Q_2 in Fig. 4(a)) only show individual trails of the analytics process at the snapshot level. The outsider issued a summarization query, Query 3 (Q_3), by specifying the aggregation over three types of vertices (viewing Alice and Bob as an abstract team member, ignoring details of files and activities), and defining the provenance meanings as a 1-hop neighborhood. The system merged Q_1 and Q_2 into a summary graph. In Fig. 4(b), the vertices suffixed name with provenance types to show alternative generation process, while edges are labeled with their frequency of appearance among segments. The query issuer would vary the summary conditions at different resolutions.

See [31] for a more in-depth discussion, including new query execution techniques for such queries.

6 Related Work

There has been much work on scientific workflow systems over the years, with some of the prominent systems being Kepler, Taverna, Galaxy, iPlant, VisTrails, Chimera, Pegasus, to name a few[2]. These systems often center around creating, automating, and monitoring a well-defined workflow or data analysis pipeline. But they cannot easily handle fast-changing pipelines, and typically are not suitable for ad hoc collaborative data science workflows where clear established pipelines may not exist except in the final, stable versions. Moreover, these systems typically do not support the entire range of tools or systems that the users may want to use, they impose a high overhead on the user time and can substantially increase the development time, and often require using specific computational environment. Further, many of these systems require centralized storage and computation, which may not be an option for large datasets.

Many users find **version control systems** (e.g., `git`, `svn`) and related hosted platforms (e.g., `GitHub`, `GitLab`) more appropriate for their day-to-day needs. However, these systems are typically too "low-level", and don't support capturing higher-level workflows or provenance information. The versioning API supported by these systems is based on a notion of files, and is not capable of allowing data researchers to reason about data contained within versions and the relationships between the versions in a holistic manner. PROVDB can be seen as providing rich introspection and querying capabilities those systems lack.

Recently, there is emerging interest in developing systems for managing different aspects in the modeling lifecycle, such as building modeling lifecycle platforms [11], accelerating iterative modeling process [45], managing developed models [33, 44], organizing lifecycle provenance and metadata [21, 30, 41], auto-selecting models [27], hosting and discovering reference models [32, 43], and assisting collaboration [24]. Issues of querying evolving and verbose provenance effectively are typically not considered in that work.

There has also been much work on **provenance**, with increasing interest in the recent years. The work in that space can be roughly categorized in two types: data provenance (aka fine-granularity) and workflow provenance (aka coarse-granularity). Data provenance is discussed in dataflow systems, such as RDBMS, Pig Latin, and Spark [3, 19, 23], while workflow provenance studies address complex interactions among high-level conceptual components in various computational tasks, such as scientific workflows, business processes, and cybersecurity [9, 12, 20, 20]. Unlike retrospective query facilities in scientific workflow provenance systems [20], their processes are predefined in *workflow skeletons*, and multiple executions generate different instance-level

---

[2]We omit the citations for brevity; a more detailed discussion with citations can be found in [30, 31]

provenance *run graphs* and have clear boundaries. Taking advantages of the skeleton, there are lines of research for advanced ancestry query processing, such as defining user views over such skeleton to aid queries on verbose run graphs [16], executing reachability query on the run graphs efficiently [8], storing run graphs generated by the skeletons compactly [5], and using visualization as examples to ease query construction [10].

Most relevant work is querying evolving script provenance [39,40]. Because script executions form clear run graph boundary, query facilities to visualize and difference execution run graphs are proposed. In our context, as there are no clear boundaries of run graphs, it is crucial to design query facilities allowing the user to express the logical run graph segments and specify the boundary conditions first. Our method can also be applied on script provenance by segmenting within and summarizing across evolving run graphs.

7 Conclusion

In this paper, we presented our approach to simplify lifecycle management of ad hoc, collaborative analysis workflows that are becoming prevalent in most application domains today. Our system, PROVDB, is based on the premise that a large amount of provenance and metadata information can be captured passively, and analyzing that information in novel ways can immensely simplify the day-to-day processes undertaken by data analysts. 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. In particular, there are many potential ways to use the rich provenance and context metadata that PROVDB collects to answer explanation queries to understand the origins of a piece of data or introspection queries to identify practices like *p-value hacking*. Continuous monitoring of this data can also help identify issues during deployment of data science pipelines (e.g., *concept drifts*). In addition, the large volumes of versioned provenance information requires development of new graph data management systems, including new graph query operators, that we are actively investigating in ongoing work.

References

[1] Provenance Challenge. `http://twiki.ipaw.info`. Accessed: 2017-07.

[2] R. Abreu, D. Archer, E. Chapman, J. Cheney, H. Eldardiry, and A. Gascón. Provenance segmentation. In *8th Workshop on the Theory and Practice of Provenance (TaPP)*, 2016.

[3] 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.

[4] M. K. Anand, S. Bowers, and B. Ludäscher. Techniques for efficiently querying scientific workflow provenance graphs. In *EDBT*, 2010.

[5] M. K. Anand, S. Bowers, T. M. McPhillips, and B. Ludäscher. Efficient provenance storage over nested data collections. In *EDBT*, 2009.

[6] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, and D. Vrgoc. Foundations of modern query languages for graph databases. *ACM Computing Surveys*, 50(5):68:1–68:40, 2017.

[7] P. B. Baeza. Querying graph databases. In *PODS*, 2013.

[8] Z. Bao, S. B. Davidson, S. Khanna, and S. Roy. An optimal labeling scheme for workflow provenance using skeleton labels. In *SIGMOD*, 2010.

[9] A. M. Bates, D. Tian, K. R. B. Butler, and T. Moyer. Trustworthy whole-system provenance for the linux kernel. In *24th USENIX Security Symposium*, 2015.

[10] L. Bavoil, S. P. Callahan, C. E. Scheidegger, H. T. Vo, P. Crossno, C. T. Silva, and J. Freire. Vistrails: Enabling interactive multiple-view visualizations. In *16th IEEE Visualization Conference (VIS)*, 2005.

[11] D. Baylor et al. TFX: A tensorflow-based production-scale machine learning platform. In *KDD*, 2017.

[12] C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying business processes. In *VLDB*, 2006.

[13] A. P. Bhardwaj, S. Bhattacherjee, A. Chavan, A. Deshpande, A. Elmore, S. Madden, and A. Parameswaran. DataHub: Collaborative Data Science & Dataset Version Management at Scale. *CIDR*, 2015.

[14] S. Bhattacherjee, A. Chavan, S. Huang, A. Deshpande, and A. Parameswaran. Principles of Dataset Versioning: Exploring the Recreation/Storage Tradeoff. *PVLDB*, 2015.

[15] S. Bhattacherjee, A. Deshpande. Storing and querying versioned documents in the cloud. In *ICDE*, 2018.

[16] O. Biton, S. C. Boulakia, S. B. Davidson, and C. S. Hara. Querying and managing provenance through user views in scientific workflows. In *ICDE*, 2008.

[17] A. Chavan and A. Deshpande. DEX: query execution in a delta-based storage system. In *SIGMOD*, 2017.

[18] A. Chavan, S. Huang, A. Deshpande, A. Elmore, S. Madden, and A. Parameswaran. Towards a Unified Query Language for Provenance and Versioning. *Theory and Practice of Provenance (TaPP)*, 2015.

[19] J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in databases: Why, how, and where. *FnT*, 2009.

[20] J. Freire, D. Koop, E. Santos, and C. T. Silva. Provenance for computational tasks: A survey. *Computing in Science & Engineering*, 2008.

[21] J. M. Hellerstein, V. Sreekanti, J. E. Gonzalez, J. Dalton, A. Dey, S. Nag, K. Ramachandran, S. Arora, A. Bhattacharyya, S. Das, M. Donsky, G. Fierro, C. She, C. Steinbach, V. Subramanian, and E. Sun. Ground: A data context service. In *CIDR*, 2017.

[22] 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 *Intl. Provenance and Annotation Workshop (IPAW)*, 2010.

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

[24] E. Kandogan, M. Roth, P. M. Schwarz, J. Hui, I. Terrizzano, C. Christodoulakis, and R. J. Miller. Labbook: Metadata-driven social collaborative data analysis. In *IEEE International Conference on Big Data*, 2015.

[25] R. H. Katz. Toward a unified framework for version modeling in engineering databases. *ACM Computing Surveys (CSUR)*, 22(4):375–409, 1990.

[26] A. Khan, S. S. Bhowmick, and F. Bonchi. Summarizing static and dynamic big graphs. *PVLDB*, 2017.

[27] A. Kumar, R. McCann, J. F. Naughton, and J. M. Patel. Model selection management systems: The next frontier of advanced analytics. *SIGMOD Record*, 44(4):17–22, 2015.

[28] M. Maddox, D. Goehring, A. J. Elmore, S. Madden, A. G. Parameswaran, and A. Deshpande. Decibel: The relational dataset branching system. *PVLDB*, 9(9):624–635, 2016.

[29] R. Mavlyutov, C. Curino, B. Asipov, and P. Cudré-Mauroux. Dependency-driven analytics: A compass for uncharted data oceans. In *CIDR*, 2017.

[30] H. Miao, A. Chavan, and A. Deshpande. ProvDB: lifecycle management of collaborative analysis workflows. In *2nd Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD*, 2017.

[31] H. Miao and A. Deshpande. Understanding data science lifecycle provenance via graph segmentation and summarization. CoRR abs/1810.04599, 2018.

[32] H. Miao, A. Li, L. S. Davis, and A. Deshpande. On model discovery for hosted data science projects. In *Workshop on Data Management for End-To-End Machine Learning, DEEM@SIGMOD*, 2017.

[33] H. Miao, A. Li, L. S. Davis, and A. Deshpande. Towards unified data and lifecycle management for deep learning. In *ICDE*, 2017.

[34] P. Missier, J. Bryans, C. Gamble, V. Curcin, and R. Dánger. ProvAbs: Model, policy, and tooling for abstracting PROV graphs. In *Intl. Provenance and Annotation Workshop (IPAW)*, 2014.

[35] P. Missier and L. Moreau. PROV-dm: The PROV data model. W3C recommendation, W3C, 2013. http://www.w3.org/TR/2013/REC-prov-dm-20130430/.

[36] L. Moreau. Aggregation by provenance types: A technique for summarising provenance graphs. In *Proceedings Graphs as Models, GaM@ETAPS 2015, London, UK, 11-12 April 2015.*, pages 129–144, 2015.

[37] L. Moreau and P. Groth. PROV-overview. W3C note, W3C, 2013. http://www.w3.org/TR/2013/NOTE-prov-overview-20130430/.

[38] K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference*, 2006.

[39] L. Murta, V. Braganholo, F. Chirigati, D. Koop, and J. Freire. noworkflow: Capturing and analyzing provenance of scripts. In *International Provenance and Annotation Workshop (IPAW)*, 2014.

[40] J. F. Pimentel, J. Freire, V. Braganholo, and L. Murta. Tracking and analyzing the evolution of provenance from scripts. In *International Provenance and Annotation Workshop (IPAW)*, 2016.

[41] S. Schelter, J. Boese, J. Kirschnick, T. Klein, and S. Seufert. Automatically tracking metadata and provenance of machine learning experiments. In *NIPS Workshop on ML Systems (LearningSys)*, 2017.

[42] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi. PGQL: a property graph query language. In *Proc. of the Intl. Workshop on Graph Data Management Experiences and Systems (GRADES)*, 2016.

[43] J. Vanschoren, J. N. van Rijn, B. Bischl, and L. Torgo. Openml: networked science in machine learning. *SIGKDD Explorations*, 15(2):49–60, 2013.

[44] 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 *Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD*, 2016.

[45] C. Zhang, A. Kumar, and C. Ré. Materialization optimizations for feature selection workloads. *ACM Transactions on Database Systems (TODS)*, 41(1):2, 2016.