Demonstration of Geyser: Provenance Extraction and Applications over Data Science Scripts

Fotis Psallidas, Megan Eileen Leszczynski[1], Mohammad Hossein Namaki, Avrilia Floratou, Ashvin Agrawal, Konstantinos Karanasos[2], Subru Krishnan, Pavle Subotić, Markus Weimer, Yinghui Wu[3], Yiwen Zhu
first.last@microsoft.com mleszczy@stanford.edu[1] kkaranasos@meta.com[2] yxw1650@case.edu[3]
Microsoft Stanford University[1] Meta[2] Case Western Reserve University[3]

ABSTRACT

As enterprises have started developing and deploying complicated data science pipelines at scale, the need for robust mechanisms that enable compliance, security, explainability, and fairness has become more pronounced. In this paper, we present Geyser, an extensible provenance system for data science workloads that can be used as a foundation for enterprise-grade data science. Our system supports a wide range of pipelines and applications by maintaining a knowledge base of data science APIs, enabling static and dynamic provenance extraction, and supporting various storage mechanisms. We demonstrate the wide applicability of the system using various industrial applications such as provenance extraction, model compliance, model linting, model versioning, and poisoning detection. A video of the demo is available at https://aka.ms/geyserdemo.

ACM Reference Format:
Fotis Psallidas, Megan Eileen Leszczynski, Mohammad Hossein Namaki, Avrilia Floratou, Ashvin Agrawal, Konstantinos Karanasos, Subru Krishnan, Pavle Subotić, Markus Weimer, Yinghui Wu, Yiwen Zhu. 2023. Demonstration of Geyser: Provenance Extraction and Applications over Data Science Scripts. In *Companion of the 2023 International Conference on Management of Data (SIGMOD-Companion '23), June 18–23, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3555041.3589717

1 INTRODUCTION

Provenance extraction has become a critical step towards enterprise-grade data science, enabling compliance auditing, security, model debuggability, reproducibility, explainability, and fairness applications. Provenance broadly encodes input-output derivation relationships between datasets across workflows. It provides a clear and transparent record of the steps performed during the analysis process. In the context of data science, examples of such relationships may include a model being trained using a group of columns from an input table, any transformation applied, or a model making a prediction given an input record.

Unfortunately, extracting provenance over data science workflows is challenging primarily due to two reasons. First, as opposed to SQL, which consists of a fixed set of well-defined operators (i.e., type of input/output and semantics of each operator), data science pipelines invoke a variety of external libraries that contain a plethora of operations [15]. Without systematically encoding the semantics of these operations (e.g., the "fit" function produces an ML model), the output of provenance extractors will not be meaningful and thus cannot be leveraged by real-world applications.

Second, provenance applications have different provenance extraction requirements. These requirements may vary in terms of fidelity (fine- vs. coarse-grained provenance) or storage (volatile vs. non-volatile and optimized for read, write, or both). For example, a model debugging application might require provenance to be stored in memory so that the information can be used immediately, whereas an auditing application might use provenance information only once per year, and thus storing it at a blob storage would be a better option. We thus need to design extensible provenance systems that (1) take into account the semantics of the data science workflows to produce meaningful output and (2) support various provenance extraction methods to better meet application needs.

In this direction, we introduced Vamsa [12], our early prototype for knowledge base (KB) driven provenance extraction. Vamsa takes as input (1) a Python script and a KB with semantic information of function calls (e.g., `pandas.read_csv` outputs a dataframe, while `sklearn.fit` outputs a trained ML model), (2) ways to encode provenance information (e.g., capture dependencies between ML models and input CSV files or/and dependencies between columns of CSV files and model features), and (3) where and how to store the information (e.g., to a remote catalog or in a local memory). It then extracts provenance through a KB-guided, static dependency analysis on the Python script. By analyzing scripts statically, Vamsa is able to scale provenance extraction to a large volume of data science pipelines. Static provenance extraction, however, is often imprecise and limited to coarse-grained provenance due to no access to runtime information (i.e., dynamic control and data flows complicate correct extraction), as we elaborate further in Section 2.

To this end, in this demonstration paper, we extend Vamsa to access runtime information and enable *dynamic* provenance extraction in a KB-driven way. The resulting system, namely Geyser, extends the functionality of Vamsa to also perform dynamic provenance extraction and capture (1) coarse-grained provenance precisely as well as (2) fine-grained provenance information. To demonstrate applications on top of Geyser, we also introduce provenance storage and querying capabilities that adhere to well-established semantics [4, 6, 9, 14]. Finally, we discuss our demonstration plans to (1) show applications of Geyser including (static and dynamic) provenance extraction, compliance testing, model linting, versioning, and poisoning detection; and (2) enable participants to interact with Geyser and its applications through Jupyter notebooks.

SIGMOD-Companion '23, June 18–23, 2023, Seattle, WA, USA

ACM ISBN 978-1-4503-9507-6/23/06…$15.00
https://doi.org/10.1145/3555041.3589717

Listing 1: Running Example

```
1.  from sklearn import tree
2.  import sklearn.model_selection import train_test_split
3.  import pandas as pd
    # loading patient data to dataframe
4.  df = pd.read_csv("heart_disease.csv")
    # specifying the set of features and ground truth
5.  df = df.drop(columns=df.columns[0:3], axis=1)
6.  train_x = df.drop(['ID', 'SSN', 'hospitalid'], axis=1)
7.  train_y = df['Target']
    # splitting data to train and evaluation sets
8.  train_x2, val_x2, train_y2, val_y2 = train_test_split(
                    train_x, train_y, test_size=0.2)
    # initializing and training the model
9.  clf = tree.DecisionTreeClassifier(random_state=1234)
10. clf.fit(train_x2, train_y2)
```

2 RUNNING EXAMPLE

To better explain the system designs of Vamsa and Geyser and their applications, we use a Python script (that was used for the Kaggle Heart Disease Competition [8]) as our running example (Listing 1). In this script, a `DecisionTreeClassifier` model is trained (Lines 9-10) using the heart_disease.csv dataset (specified in Line 4). The input features for the model are based on all columns of the dataset except the first three (`drop` call in Line 5) and the `ID`, `SSN`, and `hospitalid` columns (`drop` call in Line 6). The target label is based on the `Target` column of the input dataset (Line 7).

To illustrate the main limitation of static extractors, note that the `Target` column is used as a label in our example, but it is not explicitly dropped from the feature set, potentially leading to target leakage in ML training. With static provenance extraction, we cannot infer if the `Target` column was dropped from the feature set (`Target` may have been dropped by the `drop` call in Line 5).

To further highlight scenarios when this limitation arises in practice—hence, showcasing the importance of dynamic provenance extraction—we will extend this script during our demonstration to (1) introduce a filtering condition on age (i.e., `df[df.age >= X]`) and convert dataframes to numpy arrays before training and (2) get as input the path to the CSV file and the bound X on age as command line arguments. The first extension is useful for demonstrating how we capture and use fine-grained provenance in data science pipelines. The second extension is intended to show the importance of runtime information (command line arguments in our example) on inferring correct and complete provenance graphs.

3 SYSTEM OVERVIEW

In this section, we start by briefly reviewing the architecture of Vamsa, our KB-driven static provenance extractor. Then, we introduce Geyser, showing how we extended Vamsa to further extract dynamic provenance information in a KB-driven fashion. Finally, to support provenance applications, we discuss how we introduced provenance storage and querying capabilities in Geyser.

3.1 Static Extractor

Vamsa takes a Python script as input and statically analyzes it to extract provenance information (see also Figure 1). More specifically, Vamsa first compiles the Python script to a dataflow through its Derivation Extractor component. The dataflow is expressed in an in-house workflow-based intermediate representation (WIR) that is language-agnostic. At its core, the Derivation Extractor first encodes individual statements as quadruples (caller, operation, inputs, outputs) that we refer to as provenance relationships (PRs).

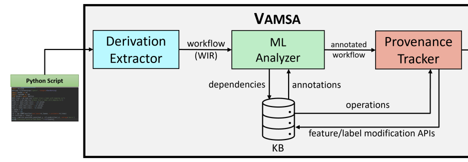

Figure 1: Vamsa (Static, KB-driven, provenance extractor).

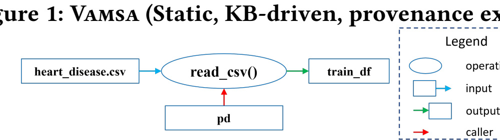

Figure 2: Example provenance relationship (PR) in our low-level, worklow-based intermediate representation.

Note that these provenance relationships are at the dataflow level and not the ML-related ones we aim to extract. For instance, for the statement `df = pd.read_csv('heart_disease.csv')`, the quadruple extracted is (caller=pd, operation=read_csv, input='heart_disease.csv', output=df). Such quadruples are then encoded in a graph form, as shown in Figure 2. Finally, based on transitivity, the Derivation Extractor connects the graphs from all PRs to form the WIR for the whole script.

The dataflow expressed in our WIR is then input to the KB-based Annotator that annotates nodes and edges of the underlying graphs with annotations from our KB. More specifically, the KB-based Annotator navigates the graph by visiting PRs starting from the PRs generated for `import` statements. For each visited PR, it searches in the KB for possible annotations and, if found, annotates the components of the PR (i.e., caller, operation, output, and input). Considering our `read_csv` example, the KB may contain that the output is a dataframe and the input is a path to a CSV file. The KB-based Annotator, then, will annotate the input/output nodes accordingly. Finally, for each visited PR, the KB-based Annotator will navigate forward based on the output(s) of the PR, but also backward through its inputs that were just annotated, if any.

The end result of the KB-based Annotator is the WIR annotated with information from the KB. This annotated WIR is then input to the Provenance Tracker that is responsible for extracting provenance information. At its core, Provenance Tracker first identifies nodes in the WIR corresponding to models and datasets. These two sets then serve, respectively, as sinks and sources in the WIR that the Provenance Tracker navigates to identify which dataset has contributed to which model: an identified path between a dataset and a model results in creating a (dataset ↔ model) provenance relationship. Furthermore, by analyzing operations in the path between models and datasets, Provenance Tracker infers which columns from the input have contributed to which features of the model (if any); hence, extracting (dataset column ↔ model feature) provenance relationships. Finally, columns are grouped into inclusion or exclusion sets based on whether they contribute or not, respectively, to a feature or label.

3.2 Dynamic Extractor

The key observation to extend our KB-driven approach to extract dynamic provenance information is that we still need to operate on the annotated WIR, albeit with the aim of dynamic dataflow analysis. In fact, the main distinction is that dynamic provenance extractors must observe runtime information (e.g., variable values,

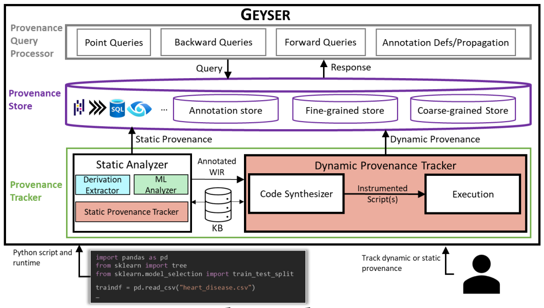

Figure 3: GEYSER: An end-to-end provenance management system for data science projects.

branches taken at runtime, records from files, or model weights) for situations where static extractors cannot make accurate decisions.

To perform dynamic provenance extraction, then, GEYSER takes a PYTHON script as input and generates its annotated WIR using the DERIVATION EXTRACTOR and KB-BASED ANNOTATOR of VAMSA (see also Figure 3). The annotated WIR and the original script are then input into the CODE SYNTHESIZER component that is responsible for injecting code in the PYTHON script to extract provenance information as part of code execution—hence, capturing dynamic provenance information. Finally, the output of the CODE SYNTHESIZER is an instrumented PYTHON script that gets executed to result in the execution of the original PYTHON script along with the extraction of dynamic provenance information.

There are two major challenges that GEYSER needs to address throughout this process: WIR verbosity and code injection.

Regarding the WIR verbosity, consider filtering records with age less than 40 in our example (`df = df[df.age >= 40]`) and later projecting the `Target` column to set the ground truth. The WIR for these steps is shown in Figure 4 (solid box). Even for these simple steps, the WIR contains many operations and temporary variables.

From a static provenance extraction perspective, this unpacking of operations to long chains is (1) necessary—because the KB may contain entries for fine-grained operations (e.g., index) and (2) convenient—because finding connections between (models ↔ datasets) or (columns ↔ features) rely only on navigating inputs and outputs of PRs in the WIR (without requiring the knowledge for connections across PRs). From a dynamic provenance extraction perspective, however, and especially for cases when we want to extract fine-grained provenance, the WIR verbosity is unnecessary and inconvenient: typically, capturing fine-grained provenance is expressed for logical operators (e.g., selection, group-by aggregation, matrix transpose) that can span multiple nodes and edges in the WIR. As such, attempting to extract such type of provenance directly on the WIR would result in identifying subgraphs of the WIR with the semantics of logical operators first. However, this is a laborious and error-prone process, and GEYSER aims to automate it.

To address this problem, the CODE SYNTHESIZER performs a condensing step to identify subgraphs of the WIR and expose them along with the semantics of logical operators. Importantly, this condensing step is driven by our KB because, over time, we expect (1) provenance to be extracted for more logical operators than the ones we currently anticipate and (2) each logical operator may have multiple ways of getting expressed in our WIR (e.g., a selection can be either a dataframe selection or an if condition in a loop).

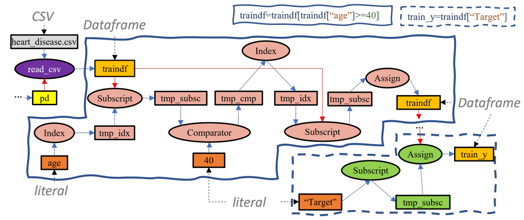

Figure 4: Example WIR highlighting subgraphs for filtering on age (solid box) and projection on `Target` (dashed box).

In more detail, recall that the input to the CODE SYNTHESIZER is the annotated WIR which maintains information for the types of inputs and outputs of each operation in the WIR. For our example, these annotations are shown in italics in Figure 4. The types of inputs and outputs introduce boundaries in our WIR that the CODE SYNTHESIZER uses to identify subgraphs of interest. The premise is that these subgraphs could correspond to higher-level logical operators (e.g., filter and projection in our example in Figure 4) for which we aim to track provenance. To perform dynamic provenance extraction, we then maintain in the KB a set of subgraph patterns and corresponding provenance tracking code to inject. The CODE SYNTHESIZER then matches identified subgraphs in the WIR with patterns stored in the KB and, in case a pattern was found, proceeds with the corresponding code injection. In this way, what type of provenance is extracted and how provenance is stored is entirely KB-driven—which provides us with the necessary extensibility we aim for. In fact, this design allows us to introduce in our KB both traditional provenance extraction techniques [6, 15], and latest advances in the data science domain [3, 5, 10] in a principled way.

In particular, the GEYSER KB currently supports coarse- and fine-grained provenance for ~358 and ~56 functions, respectively (including functions for relational/dataframe, linear algebra, and model training/inference operations). With this support, GEYSER has reached $>98\%$ precision and recall (see [12] for a definition of these metrics) on a collection of ~100 Kaggle and internal scripts—outperforming VAMSA by up to ~80% in precision and recall.

Finally, to address the second challenge (code injection), we altered the DERIVATION EXTRACTOR to perform location tracking between nodes in the WIR and the AST ones they originated from. Then, we created a shim layer that allows code to be injected before or after AST nodes. The final instrumented script is the result of compiling the altered AST back to PYTHON.

3.3 Storage and Querying

We conclude our discussion on GEYSER, by describing its provenance storage and querying layers that facilitate the specification of provenance extraction and upstream applications, respectively.

Regarding provenance storage, GEYSER exposes write APIs following well-established practices from provenance research on provenance data models and provenance storage backends for both fine- and coarse-grained provenance graphs. Regarding fine-grained provenance models, we expose both normalized and denormalized ones [6, 14]. Regarding coarse-grained models, we expose both Apache ATLAS- and W3C PROV-based type systems [1, 11] for encoding data models (i.e., types of processes and datasets of the provenance graphs). Furthermore, our APIs expose configurable write interfaces to select the backend storage. Currently, we support in-memory, Apache Arrow, database, and blob storage backends—aiming to provide broad coverage on application requirements: from applications that require immediate and fast access to provenance storage (e.g., debugging) to applications with long-term storage requirements (e.g., end-of-year auditing).

Regarding provenance querying, recall that provenance is a graph, and, as such, we expose point, backward, and forward queries following querying models [7, 9]. Furthermore, provenance graphs are also suitable for propagating annotations. In this direction, we expose interfaces that allow applications to define and propagate annotations through provenance graphs. For fine-grained provenance graphs, where operations are relational, we expose well-established semirings [9] for annotation propagation resolution. For coarse-grained provenance graphs, we follow the annotation semantics provided by Apache ATLAS and W3C PROV data models.

4 DEMONSTRATION

For our demonstration, we first show how Geyser extracts static and dynamic provenance. Then, we showcase the functionality of four applications that we built on top of Geyser: compliance, model linting, versioning, and poisoning detection. Geyser is implemented in Python, and we run our demonstration on Jupyter notebooks and connected services such as Microsoft Purview. We also welcome participants to interact with Geyser through these notebooks. A video of the demo is available at https://aka.ms/geyserdemo.

Provenance Extraction. To start our demonstration, we show the provenance information that Geyser extracts (both static and dynamic). As an input Python script, we use our running example (with modifications described in Section 2 for the dynamic case). Furthermore, to showcase the provenance stores we support (by means of supporting different provenance storage requirements), we store the coarse-grained provenance information both in-memory and in Microsoft Purview, whereas fine-grained provenance is stored in Apache Arrow (locally and in Azure Blob Storage).

Compliance. We then first use the extracted provenance information to determine whether a model relies on sensitive data (i.e., Personal Identifiable Information). For our demonstration, using the extracted coarse-grained provenance information (i.e., features of the model depending on columns of `heart_disease.csv` file), we search in Microsoft Purview for the dataset `heart_disease.csv` (using a point query through Geyser's query layer). We have pre-populated Microsoft Purview with the provenance information of `heart_disease.csv`. In particular, we assume this CSV file has been generated from a database table through an export SQL query. Furthermore, note that the `heart_disease.csv` contains an SSN column that, in our setup, originates from a database column classified as PII. Hence, by dropping or not the SSN column in our example Python script, we fail or not the compliance rule.

Model Linting. Similarly to compliance testing, we perform checks for model linting. In particular, recall from Section 2 that the `Target` column is set as the label, yet it is not apparent whether it is used in the feature set. Using the static provenance information, model linting then triggers a corresponding warning (i.e., indicating there may or may not be a problem). Using dynamic provenance information, however, we showcase that we can figure out whether `Target` is used as a feature and trigger an error or not.

Versioning. Furthermore, we use provenance extracted across runs to explain why the resulting models are different or identical. For our demonstration, we run our example Python script by changing the filter on age and modifying the input dataset. We then analyze differences between both coarse- and fine-grained provenance to determine why a model (1) is different (e.g., the input file changed) or (2) same (e.g., filter change → no selectivity change) across runs.

Poisoning Detection. Finally, we use fine-grained provenance to detect poisonous sources using a poisoning detection algorithm [2, 13]. The algorithm takes as input annotations of each row (i.e., its source and, optionally, if it is trustworthy) of the feature matrix that is input to model training. The algorithm then tests whether the model performs similarly for each source; the performance for poisonous sources is expected to differ. For our setup, we need to construct these annotations based on information available in the input file. Hence, fine-grained provenance is critical. In particular, `heart_disease.csv` contains patient-related information (one patient per record), and each such record originates from a hospital. We designate one hospital to be a poisonous source (we altered labels for this source), another to be trusted, and two to be unknown. To construct the annotations, we backward trace from each row in the feature matrix to the input records of the CSV file—to discover the source hospitals. Then, we forward trace in the opposite direction to propagate annotations for the hospital we already trust. With these trace operations, the annotations are now constructed and are input to the poisoning detection algorithm.

REFERENCES

[1] 2022. Apache Atlas. https://atlas.apache.org/
[2] Nathalie Baracaldo, Bryant Chen, Heiko Ludwig, Amir Safavi, and Rui Zhang. 2018. Detecting Poisoning Attacks on Machine Learning in IoT Environments. In *ICIOT*. 57–64.
[3] Adriane Chapman, Luca Lauro, Paolo Missier, and Riccardo Torlone. 2022. DPDS: assisting data science with data provenance. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3614–3617.
[4] Adriane P Chapman, Hosagrahar V Jagadish, and Prakash Ramanan. 2008. Efficient provenance storage. In *SIGMOD '18*. 993–1006.
[5] Nachiket Deo, Boris Glavic, and Oliver Kennedy. 2022. Runtime Provenance Refinement for Notebooks. In *TaPP '22*.
[6] Boris Glavic and Gustavo Alonso. 2009. Perm: Processing provenance and data on the same data model through query rewriting. In *ICDE*.
[7] Robert Ikeda. 2012. *Provenance In Data-Oriented Workflows*. Ph. D. Dissertation. Stanford University.
[8] Kaggle. 2022. *Kaggle Heart Disease Competition*. Retrieved December 14, 2022 from https://www.kaggle.com/datasets/rishidamarla/heart-disease-prediction
[9] Grigoris Karvounarakis, Zachary G Ives, and Val Tannen. 2010. Querying data provenance. In *SIGMOD '10*. 951–962.
[10] Stephen Macke, Hongpu Gong, Doris Jung-Lin Lee, Andrew Head, Doris Xin, and Aditya Parameswaran. 2021. Fine-grained lineage for safer notebook interactions. *Proceedings of the VLDB Endowment* 14, 6 (2021), 1093–1101.
[11] Paolo Missier, Khalid Belhajjame, and James Cheney. 2013. The W3C PROV family of specifications for modelling provenance metadata. In *EDBT '13*. 773–776.
[12] Mohammad Hossein Namaki, Avrilia Floratou, Fotis Psallidas, Subru Krishnan, Ashvin Agrawal, Yinghui Wu, Yiwen Zhu, and Markus Weimer. 2020. Vamsa: Automated Provenance Tracking in Data Science Scripts. In *SIGKDD '20*. 1542–1551.
[13] Maria-Irina Nicolae, Mathieu Sinn, Minh Ngoc Tran, Beat Buesser, Ambrish Rawat, Martin Wistuba, Valentina Zantedeschi, Nathalie Baracaldo, Bryant Chen, Heiko Ludwig, Ian Molloy, and Ben Edwards. 2018. Adversarial Robustness Toolbox v1.2.0. *CoRR* 1807.01069 (2018). https://arxiv.org/pdf/1807.01069
[14] Fotis Psallidas and Eugene Wu. 2019. Smoke: Fine-Grained Lineage at Interactive Speed. *PVLDB* 11, 6 (jan 2019), 719–732.
[15] Fotis Psallidas, Yiwen Zhu, Bojan Karlas, Jordan Henkel, Matteo Interlandi, Subru Krishnan, Brian Kroth, Venkatesh Emani, Wentao Wu, Ce Zhang, Markus Weimer, Avrilia Floratou, Carlo Curino, and Konstantinos Karanasos. 2022. Data Science Through the Looking Glass: Analysis of Millions of GitHub Notebooks and ML.NET Pipelines. *SIGMOD Record* 51, 2 (jul 2022), 30–37.