
SUPPORTING BETTER INSIGHTS OF DATA SCIENCE PIPELINES WITH FINE-GRAINED PROVENANCE

Adriane Chapman

University of Southampton, UK
Adriane.Chapman@soton.ac.uk

Luca Lauro

Università Roma Tre, Italy
luca.lauro@uniroma3.it

Paolo Missier

Newcastle University, UK
lpaolo.missier@ncl.ac.uk

Riccardo Torlone

Università Roma Tre, Italy
riccardo.torlone@uniroma3.it

ABSTRACT

Successful data-driven science requires complex data engineering pipelines to clean, transform and alter data in preparation for machine learning, and robust results can only be achieved when each step in the pipeline can be justified, and its effect on the data explained. In this framework, our aim is to provide data scientists with facilities to gain an in-depth understanding of how each step in the pipeline affects the data, from the raw input to training sets ready to be used for learning. Starting from an extensible set of data preparation operators commonly used within a data science setting, in this work we present a provenance management infrastructure for generating, storing, and querying very granular accounts of data transformations, at the level of individual elements within datasets whenever possible. Then, from the formal definition of a core set of data science preprocessing operators, we derive a *provenance semantics* embodied by a collection of templates expressed in PROV, a standard model for data provenance. Using those templates as a reference, our provenance generation algorithm generalises to any operator with observable input/output pairs. We provide a prototype implementation of an application-level provenance capture library to produce, in a semi-automatic way, complete provenance documents that account for the entire pipeline. We report on that reference implementations ability to capture provenance in real ML benchmark pipelines and over TCP-DI synthetic data. We finally show how the collected provenance can be used to answer a suite of provenance benchmark queries that underpin some common pipeline inspection questions, as expressed on the Data Science Stack Exchange.

Keywords Provenance, Data Science, Data Preparation, Preprocessing

1 Introduction

Dataset selection and data wrangling pipelines are integral to applied Data Science workflows. These typically culminate in the generation of predictive models for a broad range of data types and application domains through training. A number of critical choices are made when these pipelines are designed, starting with the choice of which datasets to include or exclude, how these should be merged [1], and which transformations are required to produce a viable training set, given a choice of target learning algorithms. The main intended consequence of these transformation pipelines is to optimise the predictive performance and generalisation characteristics of the models that are derived from the ground data. There are however also unintended consequences, as these transformations alter the representation of the domain that the learning algorithms generalise from, and they may remove or inadvertently introduce new bias in the data [2]. In turn, this may reflect on non-performance properties of the models, such as their *fairness*. The term, formally defined in terms of statistical properties of the model's predictions [3], broadly refers to the capability of a model to ensure that its predictions are not affected by an individual belonging to one of the groups defined by some sensitive attribute(s), such as sex, ethnicity, income band, etc.

Motivation. Models that are provably fair are also perceived as more trustworthy, an important feature at a time when machine learning models are increasingly used to support and complement human expert judgment, in areas where decisions have consequences on individuals as well as on businesses. Substantial recent research has produced techniques for explanation using: counterfactuals [4], local explanations [5], data [6] and meta-models [7]. While these techniques focus primarily on the model itself, relatively little work has been done into trying to explain models in terms of the transformations that occur *before* the data is used for learning. The ultimate goal of this work is to enable explanations of the effect of each transformation in a pre-processing pipeline on the data that is ultimately fed into a model [8]. As an initial step in this direction, we have developed a formal model and practical techniques for recording data derivations at the level of the atomic elements in the dataset, for a general class of data transformation operators. These derivations are a form of data provenance and are expressed using the PROV data model [9], a standard and widely adopted ontology. Data derivations form a corpus of graph-structured metadata that can be queried as a preliminary step to support user questions about model properties.

Problem scope. In this paper we focus on data transformations that are commonly found in data science processing pipelines and across application domains, and we further limit the scope to structured tabular data.¹ These steps have been systematically enumerated in multiple reviews (see e.g. [10, 11]) and include, among others: feature selection, engineering of new features; imputation of missing values, or *listwise deletion* (excluding an entire record if data is missing on any variable for that record); downsampling or upsampling of data subsets in order to achieve better balance, typically on the class labels (for classification tasks) or on the distribution of the outcome variable (for regression tasks); outlier detection and removal; smoothing and normalisation; de-duplication, as well as steps that preserve the original information but are required by some algorithms, such as “one-hot” encoding of categorical variables. A complex pipeline may include some or all of these steps, and different techniques, algorithms, and choice of algorithm-specific parameters may be available for each of them. These are often grounded in established literature but variations can be created by data scientists to suit specific needs. In this work we consider the space of all configured pipelines that can potentially be composed out of these operators.

Regarding the data that these operate on, we focus on structured two-dimensional tabular data, namely dataframes, which are commonly supported by R and python as well as by a dedicated Spark API, and excluding tensors and multi-dimensional matrices. While this is done to simplify our proof-of-concept implementation, we observe that considering higher-dimension tabular structures has practical implications as it increases the complexity of the derivations from input to output elements, however the underpinning provenance templates are fundamentally the same.

Overview of the approach. Firstly, we provide a formalisation and categorisation of a core set of these operators. Then, with each class of those operators, we associate a *provenance template* that describes the effect on the data of each operator in the class at the appropriate level of detail, i.e., on individual data elements, columns, rows, or collections of those. By mapping operators to these fundamental templates, we are then able to identify the transformation type based on observation of the operator’s input and outputs alone. By abstracting to this level, we can automatically create the appropriate provenance for an operator in a data science pipeline if it follows the pre-identified input-output patterns, even if the operator itself has never been seen before.

Contributions. Our contributions can be summarised as follows.

- A formalisation and categorisation of a core set of operators for data reduction, augmentation, transformation, and fusion that move beyond the relational algebra (Section 3), showing how common data pre-processing pipelines can be expressed as a composition of these operators.
- The semantics of the provenance that is generated for white-box transformations, as reduced to the core set of operators (Section 4).
- A method for capturing the provenance of a pipeline, based on *observing the changes to the data*, not the operator that was applied (Section 5).
- An application-level provenance capture facility for Python, underpinned by the formal model, that (i) identifies the operation under execution to capture its provenance and (ii) is backed by a Neo4J database used as a provenance store (Section 6). This new approach almost entirely removes the older requirement for pipeline designers to programmatically “drive” provenance generation, making most of the process transparent;
- Using a reference implementation, we report on: (i) the impact of adding provenance capture to real-world pipelines (Section 7.1), (ii) the ability to capture provenance in real ML benchmark pipelines and over TCP-DI synthetic data [12] (Section 7.2), (iii) a use case analysis showing that provenance queries can provide support to data scientists in the development of real-world machine-learning pipelines (Section 7.3) (iv) how data

¹However, we are not going to consider more specialised data pre-processing steps that may apply to data types such as video, audio, images, etc.

provenance collected with our approach can be inspected through user-friendly interfaces (Section 7.4), and (v) a comparison to other similar provenance capture systems (Section 7.5).

2 Models and Problem statement

2.1 Data model

The data collected for ML tasks are usually represented as tables or statistical data matrices in which columns represent specific features of a phenomenon being observed, and rows are records of data for those features describing observations of the phenomenon. To capture both formats, we will refer to these generically as *datasets*, similar in spirit to notions of ordered relations [13] and dataframes [14].

A (*dataset*) *schema* S is an array of distinct names called *features* (or *attributes*) $S = [\mathbf{a}_1, \dots, \mathbf{a}_n]$. Each feature is associated with a domain of atomic values (such as numbers, strings, and timestamps). With a little abuse of notation, hereinafter we will compare schemas using set containment over their features. A *dataset* D over a schema $S = [\mathbf{a}_1, \dots, \mathbf{a}_n]$ is an ordered collection of *rows* (or *records*) of the form: $i : (d_{i1}, \dots, d_{in})$ where i is the unique *index* of the row and each element d_{ij} (for $1 \leq j \leq n$) is either a value in the domain of the feature \mathbf{a}_j or the special symbol \perp , denoting a missing value. Row indexes can be implemented in different ways (e.g., with RID annotations [15]). We only assume here that a row of any dataset can be uniquely identified.

Given a dataset D over a schema S we denote by $D_{i\mathbf{a}}$ the value for the feature \mathbf{a} of S occurring in the i -th row of D . We also denote by D_{i*} the i -th row of D , and by $D_{*\mathbf{a}}$ the column of D associated with the feature \mathbf{a} of S .

Example 2.1 A possible dataset D over the schema $S = [Cid, Gender, Age, Zip]$ is as follows:

	<i>Cid</i>	<i>Gender</i>	<i>Age</i>	<i>Zip</i>
1	113	<i>F</i>	24	98567
2	241	<i>M</i>	28	\perp
3	375	<i>C</i>	\perp	32768
4	578	<i>F</i>	44	32768

D_{*Age} and D_{2*} denote the third column and the second row of D , respectively.

Note that, as mentioned in the introduction, in this work we focus on dataframes, which are described by a schema. Extensions to tensors and multidimensional matrices are left for future work.

2.2 Data manipulation model

A *general classification*.

As part of this work, we analyzed several packages that allow users to build data pre-processing pipelines. Table 1 contains an example overview of the available operators from the ML pipeline building tool Orange [16] and the popular SciKit packages [17]. As indicated on the left-hand side of the table, all of them can be classified into four main classes, according to the type of manipulation done on the input dataset(s) over a schema S :

- **Data reductions:** operations that take as input a dataset D on a schema S and reduce the size of D by eliminating rows (without changing S) or columns (changing S to $S' \subset S$) from D ;
- **Data augmentations:** operations that take as input a dataset D on a schema S and increase the size of D by adding rows (without changing S) or columns (changing S to $S' \supset S$) to D ;
- **Data transformations:** operations that take as input a dataset D on a schema S and, by applying suitable functions, transform (some of) the elements in D without changing its size or its schema (up to possible changes to the domain of the involved features of S)
- **Data fusions:** operations that take as input two datasets D_1 and D_2 on schema S_1 and S_2 respectively and combine them into a new dataset D on a schema S involving the features of S_1 and S_2 .

We now introduce a number of basic operators of data manipulation over datasets belonging to one of the above classes of data manipulations, as indicated in the right-hand side of Table 1. This approach is in line with the observation that most of the operations of current data exploration packages rely on a rather small subset of operators [14].

Data reductions. Two basic data reduction operators are defined over datasets. They are simple extensions of two well-known relational operators.

Table 1: Typical operations in ML pipelines of data preparation from Orange [18] and Scikit-Learn [17].

Orange3 Ex.	ScikitLearn Ex.	Category	Operator	Implementation
Feature Statistics	Feature_selection	Data reduction	Feature Selection	π_C
Select Data by Index	Dataframe op.		Instance Selection	σ_C
Select Columns	Feature_selection		Drop Columns	π_C
Select Rows	Dataframe op.		Drop Rows	σ_C
Data Sampler	Imbalanced-learn		Undersampling	σ_C
Impute	SimpleImputer	Data transformation	Imputation	$\tau_{f(X)}$
Apply Domain	FunctionTransformer		Value Transformation	$\tau_{f(X)}$
Edit Domain	Binarizer		Binarization	$\tau_{f(X)}$
Preprocess	Normalizer		Normalization	$\tau_{f(X)}$
Discretize	KBinDiscretizer		Discretization	$\tau_{f(X)}$
Feature Constructor	FunctionTransformer	Data augmentation	Space Transformation	$\pi_Z \circ \alpha_{\vec{f}(X):Y}$
Create Class	FunctionTransformer		Instance Generation	$\alpha_{X:f(Y)}^\downarrow$
Data Sampler	Imbalanced-learn		Oversampling	$\alpha_{X:f(X)}^\downarrow$
Corpus	Label Encoder		String Indexer	$\alpha_{\vec{f}(X):Y}$
Preprocess	OneHotEncoder		One-Hot Encoder	$\alpha_{\vec{f}(X):Y}$
Merge	Hstack	Data fusion	Join	\bowtie_C^t
Concatenate	Vstack		Append	\uplus

π_C : the (conditional) projection of D on a set of features of S that satisfy a boolean condition C over S , denoted by $\pi_C(D)$, is the dataset obtained from D by including only the columns $D_{*\mathbf{a}}$ of D such that \mathbf{a} is a feature of S that satisfy C ;

σ_C : the selection of D with respect to a boolean condition C over S , denoted by $\sigma_C(D)$, is the dataset obtained from D by including the rows D_{i*} of D satisfying C .

The condition of both the projection and the selection operators can refer to the values in D , as shown in the following example that uses an intuitive syntax for the condition.

Example 2.2 Consider the dataset D in Example 2.1. The result of the expression $\pi_{\{\text{features without nulls}\}}(\sigma_{\text{Age} < 30}(D))$ is the following dataset:

	<i>CId</i>	<i>Gender</i>	<i>Age</i>
1	113	F	24
2	241	M	28

Data augmentations. Two basic data augmentation operators are defined over datasets. They allow the addition of columns and rows to a dataset, respectively.

$\alpha_{\vec{f}(X):Y}$: the vertical augmentation of D to Y using a function f over a set $X = [\mathbf{a}_1 \dots \mathbf{a}_k] \subseteq S$ of features, is obtained by adding to D a new set of features $Y = [\mathbf{a}'_1 \dots \mathbf{a}'_l]$ whose new values $d_{i\mathbf{a}'_1} \dots d_{i\mathbf{a}'_l}$ for the i -th row are obtained by applying f to $d_{i\mathbf{a}_1} \dots d_{i\mathbf{a}_k}$;

$\alpha_{X:f(Y)}^\downarrow$: the horizontal augmentation of D using an aggregative function f is obtained by adding one or more new rows to D obtained by first grouping over the features in X and then, for each group, by applying f to $\pi_Y(D)$ (extending the result to S with nulls if needed). Note that horizontal augmentation generates new rows based on grouping, i.e., by X , followed by an aggregation $f(Y)$ applied to the values for Y in each group.

Example 2.3 Consider again the dataset D in Example 2.1 and the following functions: (i) f_1 , which associates the string young when age is less than 25 and the string adult otherwise, and (ii) f_2 , which computes the average of a set of numbers. Then, the expression $\alpha_{\vec{f}_1(\text{Age}):ageRange}^\downarrow(D)$ produces the following dataset:

	<i>CId</i>	<i>Gender</i>	<i>Age</i>	<i>Zip</i>	<i>ageRange</i>
1	113	F	24	98567	young
2	241	M	28	\perp	adult
3	375	C	\perp	32768	\perp
4	578	F	44	32768	adult

In expression $E_2 = \alpha_{\text{Gender}:avg(\text{Age})}^\downarrow(D)$ first group by **Gender** is computed, yielding two groups (for M and F), then $avg(\text{Age})$ is executed on each group, resulting in the new rows 5,6 in the dataframe below:

	<i>CId</i>	<i>Gender</i>	<i>Age</i>	<i>Zip</i>
1	113	<i>F</i>	24	98567
2	241	<i>M</i>	28	⊥
3	375	<i>C</i>	⊥	32768
4	578	<i>F</i>	44	32768
5	⊥	<i>F</i>	34	⊥
6	⊥	<i>M</i>	28	⊥

Note that new data can be added to a dataset using a horizontal augmentation where $X = \emptyset$, $Y = S$, and f denote the procedure for adding records (e.g., by asking them to the user). Note also that horizontal augmentation allows us to combine, in the same dataset, entities at different levels of granularity, a feature that can be very useful to a data scientist (e.g., to compute, in the example above, the mean deviation).

Data transformation. One basic data transformation operator is defined over datasets:

$\tau_{f(X)}$: the transformation of a set of features X of D using a function f is obtained by substituting each value d_{ia} with $f(d_{*a})$, for each feature a occurring in X .

Example 2.4 Let D be the dataset in Example 2.1 and f be an imputation function that associates to the \perp 's occurring in a feature a the most frequent value occurring in D_{*a} . Then, the result of the expression $\tau_{f(\text{zip})}(D)$ is the following dataset:

	<i>CId</i>	<i>Gender</i>	<i>Age</i>	<i>Zip</i>
1	113	<i>F</i>	24	98567
2	241	<i>M</i>	28	32768
3	375	<i>C</i>	⊥	32768
4	578	<i>F</i>	44	32768

Data fusion. Given D^L and D^R on schemas S^L and S^R respectively, the two basic data fusion operators *join* and *append* allow the combination of a pair of datasets.

- the *join* $D^L \bowtie_C^t D^R$ of D^L and D^R based on a boolean condition C is the dataset over $S^L \cup S^R$ obtained by applying standard join operation of type t (where t can be equal to inner, (left/right/full) outer) based on the condition C ;
- the *append* $D^L \uplus D^R$ of D^L to D^R is the dataset over $S^L \cup S^R$ obtained by appending D^L to D^R and possibly extending the result with nulls on the mismatching columns $(S^L \cup S^R) \setminus (S^L \cap S^R)$.

Example 2.5 Let D^L be the dataset in Example 2.1 (which we report here for convenience) and D^R the dataset that follows.

	<i>CId</i>	<i>Gender</i>	<i>Age</i>	<i>Zip</i>
1	113	<i>F</i>	24	98567
2	241	<i>M</i>	28	⊥
3	375	<i>C</i>	⊥	32768
4	578	<i>F</i>	44	32768

	<i>CId</i>	<i>name</i>
1	241	<i>Jim</i>
2	578	<i>Mary</i>

Then, the result of the expression $D^L \bowtie_{D^L.CId=D^R.CId}^{\text{inner}} D^R$ is the following dataset:

	<i>CId</i>	<i>Gender</i>	<i>Age</i>	<i>Zip</i>	<i>Name</i>
1	241	<i>M</i>	28	⊥	<i>Jim</i>
2	578	<i>F</i>	44	32768	<i>Mary</i>

On the other hand, the result of the expression $D^L \uplus D^R$ is the following dataset:

	<i>CId</i>	<i>Gender</i>	<i>Age</i>	<i>Zip</i>	<i>Name</i>
1	113	<i>F</i>	24	98567	⊥
2	241	<i>M</i>	28	⊥	⊥
3	375	<i>C</i>	⊥	32768	⊥
4	578	<i>F</i>	44	32768	⊥
5	241	⊥	⊥	⊥	<i>Jim</i>
6	578	⊥	⊥	⊥	<i>Mary</i>

We note that the data manipulation model presented here has some similarities with the Dataframe algebra [14]. The main difference is that we have focused on a restricted set of core operators (with some of that in [14] missing and others combined in one) with the specific goal of providing a solid basis to an effective technique for capturing data provenance of classical preprocessing operators. We point out that our algebra can be easily extended to include operators implementing other ETL/ELT-like transformations whose fine-grained provenance capture has been described elsewhere [19].

2.3 Data provenance model

The purpose of data provenance, in this setting, is to support the generation of simple explanations for the existence (or the absence) of some piece of data in the result of complex data manipulations. Along this line, we adopt as the provenance model a subset of the PROV model [20] from the W3C, a widely adopted ontology that formalises the notion of *provenance document* and which admits RDF and other serialisation formats to facilitate interoperability. The minimal elements of the model are graphically described as shown in Figure 1.

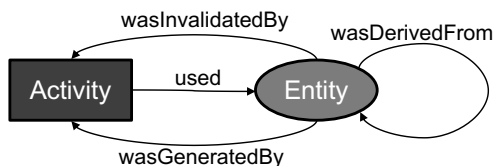


Figure 1: The core W3C PROV model.

In PROV an entity represents an element d of a dataset D and is uniquely identified by D and the coordinates of d in D (i.e., the corresponding row index and feature). An activity represents any pre-processing data manipulation that operates over datasets. For each element d in a dataset D' generated by an operation \circ over a dataset D we represent the facts that: (i) d *wasGeneratedBy* \circ , and (ii) d *wasDerivedFrom* a set of elements in D . In addition, we represent: (iii) all the elements d of D such that d *was used* by \circ and (iv) all the elements d of D such that d *wasInvalidatedBy* (i.e., deleted by) \circ (if any). Note that in PROV derivation implies usage, but the inverse is not true and this is why this notation is not redundant.

Example 2.6 Let E be the first expression in Example 2.3 and $D' = E(D)$. A fragment of the data provenance generated by this operation, for two of the dataset elements, is reported in Figure 2.

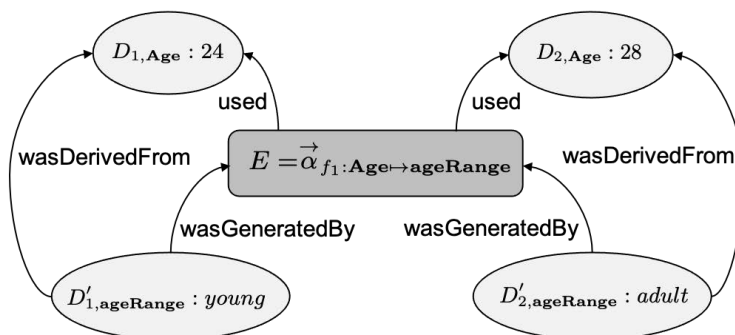


Figure 2: A fragment of provenance data for the operation in Example 2.6.

2.4 Limitations and possible extensions

The models for data representation, manipulation, and provenance generation introduced in the previous sections cover a large body of data preparation pipelines, but they are clearly not exhaustive. In particular, we have assumed that the input data is in a bi-dimensional, tabular format (e.g., csv files), with rows representing observations of some phenomena and columns representing interesting features of the observations. However, multi-dimensional data, including tensors and matrices, are common in many machine learning applications. Our model can be extended by assuming that each value is indeed a measure for a combination of features, possibly at different levels of aggregation, similar to logical multidimensional data models that have been proposed in the literature for data warehousing and OLAP (e.g., [21]).

This would also make it possible to include multi-level aggregation operations (roll-up, drill-down, slicing, and dicing) by extending the data model as described e.g. in [21]. These extensions add complexity to the resulting provenance graphs, as the derivations must be traced through multiple aggregations, however, this would not add to our conceptual framework, as these simply extend the fundamental provenance patterns used for standard one-level aggregations (see Sec.4.2.4). Supporting such extensions is therefore currently beyond the scope of our proof-of-concept implementation.

2.5 Problem Statement

We consider compositions of the operators introduced in Section 2.2 into *pipelines* that take as input a collection of datasets D_1, \dots, D_n and produce a dataset D' , denoted $D' = p(D_1, \dots, D_n)$, by applying a (partially ordered) sequence of operators. Note that E can be represented as a tree, where the internal nodes are operators and leaves are datasets. Note also that although in principle any combination is possible, in practice there are constraints on the ordering of the operators, because some operators may alter the dataset schema.

The performance of the model learned from $p(D)$ is dependent upon the operators involved in p . As the data scientist iterates over versions of the models, they may wish to inspect and understand exactly what happened at each step within the pipeline. This can be a complex manual task for any realistic pipeline.

The provenance collected by the system presented here is intended to allow the data scientist to review, understand, and debug what happened in past runs of any given pipeline. Depending on the granularity of the provenance, this can be as coarse-grained as a dataset, p was used with transformations T_1, T_2, \dots to a very fine-grained version which allows users to track individual data items as described previously.

Classic provenance queries include: Why [22], How [23] and Why Not [24]. Instances of each of these queries are shown in Table 2 as Queries 2, 3, and 7-9 respectively. In addition to these classic provenance queries, we have analyzed questions posed to the Data Science Stack Exchange (DSSE) about problems posed by users, encountered when trying to understand and debug the pipelines. An explanation of the use cases and the provenance queries in Table 2 that they relate to can be found in Table 7. Through this analysis, we have identified an additional 6 provenance queries based on the use cases from DSSE: All Transformations (1); Dataset-level Feature Operation (4); Record Operation (5); Item-level Feature Operation (6); Impact on Feature Spread (12) and Impact on Dataset Spread (13). Queries 1, 4, and 5 are similar to How-provenance but are focused only on the transformations. The difference between them is the granularity of focus - dataset, feature, record, or individual value. Queries 10 and 11 have been implemented to emphasize how provenance can help when developing a pipeline. They show what an item was and what it will be, highlighting potential errors or imperfections. Queries 12 and 13, however, present a new usage of provenance, and thus a distinctly new provenance query type.

In the DSSE use cases, it became clear that a question being asked was “what operations were performed to the data and how did those change the data profile”. This is a reasonable question as these transformations may entail unintended consequences, as they alter the representation of the domain that the learning algorithms generalize from, and they may remove or inadvertently introduce new bias in the data [2]. In turn, this may reflect on non-performance properties of the models, such as their *fairness*. Fairness, formally defined in terms of statistical properties of the model’s predictions [3], broadly refers to the capability of a model to ensure that its predictions are not affected by an individual belonging to one of the groups defined by some sensitive attribute(s), such as sex, ethnicity, income band, etc. Queries 12 and 13 provide a mechanism that computes the statistical properties of the data before and after an operation to identify when there are major shifts in distributions.

Thus, the problem within this work is to: a) define the set of operations for data manipulation available within a pipeline; b) establish a set of provenance templates that can be used to reason over and capture the provenance of these operations over the data; c) show that our approach can support typical provenance queries in an effective and scalable way.

3 Data processing operators

In this section, we illustrate a number of common data processing operators that are often used in data preparation workflows, showing how they can be suitably expressed as a composition of the basic operators introduced in Section 2.2.

3.1 Data Reduction

Feature Selection. This operation consists of selecting a set of relevant features from a given dataset and dropping the others, which are either redundant or irrelevant to the goal of the learning process.

Table 2: Provenance queries of interest to a data scientist designing a pipeline of pre-processing operations.

Id	Provenance Query	Input	Output
PQ1	All Transformations	D	Set of operations applied to D and the features they affect.
PQ2	Why-provenance	d_{ia}	The input data that influenced d_{ia} .
PQ3	How-provenance	d_{ia}	The input data and the operations that created d_{ia} .
PQ4	Dataset-level Feature Operation	D_{*a}	Set of operations that were applied to feature \mathbf{a} .
PQ5	Record Operation	D_{i*}	Set of operations that were applied to record D_{i*} .
PQ6	Item-level Feature Operation	d_{ia}	Set of operations that were applied to d_{ia} .
PQ7	Feature Invalidation	D, \mathbf{a}	The operation that deleted the feature \mathbf{a} .
PQ8	Record Invalidation	D, i	The operation that deleted the record D_{i*} .
PQ9	Item Invalidation	D, i, \mathbf{a}	The operation that deleted the item d_{ia} .
PQ10	Item History	d_{ia}	All the elements derived and that will derive from d_{ia} .
PQ11	Record History	D_{i*}	All the elements derived and that will derive from D_{i*} .
PQ12	Impact on Feature Spread	D	The change in feature spread of all operations over a feature of D .
PQ13	Impact on Dataset Spread	D	The change in dataset spread of all operations applied to D .

Feature selection over a dataset D with a schema S can be expressed by means of a simple pipeline involving only the projection operator with a condition that selects the set of features $I \subset S$ of interest:

$$FS(D) = \pi_C(D)$$

where $C = \{\mathbf{a} \in I\}$.

A special case of feature selection is an operation that drops columns with a value rate of missing values higher than a threshold t . In this case, the condition of the projection operator is more involved as it requires introspection of the dataset:

$$C = \{\mathbf{a} \in S \mid \text{count}(D_{ia} = \perp, 1 \leq i \leq n) < t\}.$$

Instance Selection. The aim of this operation is to reduce the original dataset to a manageable volume by removing certain records with the goal of improving the accuracy (and efficiency) of classification problems.

Also in this case, instance selection over a dataset D with a schema S can be expressed by means of a simple pipeline involving only the selection operator with a condition that identifies the set of relevant rows of D by means of a predicate p : $IS(D) = \sigma_C(D)$ where $C = \{D_{i*} \in S \mid p(D_{i*})\}$.

Similar to feature selection, a relevant case of instance selection drops rows with a value rate of missing values higher than a threshold t . In this case,

$$C = \{D_{i*} \in D \mid \text{count}(D_{ij} = \perp, 1 \leq j \leq m) < t\}$$

3.2 Data Transformations

By data transformation, we mean any operation on a given dataset that modifies its values with the goal of improving the quality of D and/or making the process of information extraction from D more effective. The following operator is meant to capture data transformation (DT in its generality):

$$DT(D) = \tau_{f(X)}(D)$$

where f is any scalar function that generates a new value $f(x)$ from values of feature set X of S . Several cases of transformations are common in pre-processing pipelines, as illustrated in the following.

Data repair. It is the process of replacing inconsistent data items with new values. In this case, f is a simple function that converts values and the data transformation possibly operates on the whole dataset.

Binarization. It is the process of converting numerical features to binary features. For instance, if a value for a given feature is greater than a threshold it is changed a 1, if not to 0.

Normalization. It is a scaling technique that transforms all the values of a feature so that they fall in a smaller range, such as from 0 to 1. There are many normalization techniques, such as Min-Max normalization, Z-score normalization, and Decimal scaling normalization. This operation operates on a single feature at a time

Discretization. It consists of converting or partitioning continuous features into discrete or nominal features. It performs a value transformation from categorical to numerical data.

Imputation. It is the process of replacing missing data (nulls in our data model) with valid data using a variety of statistical approaches that aim at identifying the values with the maximum likelihood.

3.3 Data augmentation

Space Transformation. This operation takes a set of features of an existing dataset and generates from them a new set of features by combining the corresponding values. Usually, the goal is to represent (a subset of) the original set of features in terms of others in order to increase the quality of learning.

The application of this operation to a dataset D over a schema S can be expressed by means of an expression involving a vertical augmentation that operates on a subset X of the features in S and produces a new set of features Y , followed by a projection operator that eliminates the features in X , thus maintaining those in $Z = (S \cup Y) - X$:

$$ST(D) = \pi_{\{features\ in\ Z\}}(\alpha_{f(X):Y}^{\rightarrow}(D))$$

Instance Generation: These operators include grouping and aggregation, and their effect is to fill regions in the domain of the problem, which does not have representative examples in original data, or to summarize large amounts of instances in fewer examples. These are also denoted *prototype generation* methods, as the artificial examples created tend to act as a representative of a region or of a subset of the original instances.

The application of this operation to a dataset D over a schema S can be expressed by means of an expression involving a horizontal augmentation that, if needed, groups over a subset X of the features in S and then apply a summary function f over another subset of S :

$$IG(D) = \alpha_{X:f(Y)}^{\downarrow}(D).$$

This operation can be preceded by a data reduction operator (a projection or a selection) to isolate the portion of the original dataset on which we intend to operate.

String Indexer. This operator encodes a feature involving strings into a feature of string indices. The indices are in $[0, numLabels)$. It is a special case of Space transformation.

One-Hot Encoder. This operation maps a feature involving strings to a set of boolean features. Specifically, it creates one column for each possible value occurring in the feature. Each new feature gets a 1 if the row contained that value and a 0 if not. It is a special case of space transformation.

3.4 Data fusion

Data preparation pipelines often require combining datasets coming from different data sources. For this reason, packages for data pre-processing are usually equipped with facilities for combining datasets in two main ways, as follows.

Data integration. It is the process of combining rows of two datasets on the basis of some common property. This can be useful when, for instance, we need to extend the features of observations of a phenomenon or objects of interest stored in a dataset D_1 (e.g., the technical information of smartphones on sale) with further features of the same observations or of the same objects gathered elsewhere stored in a dataset D_2 (e.g., the ratings of the same smartphones available on a review site). This activity can be supported by an expression involving the join operator over the datasets under consideration and can be preceded by a data reduction operator to isolate the portion X of the original dataset on which we intend to operate, as follows:

$$\pi_X(D_1 \bowtie_C^{left} D_2)$$

where C specifies the condition that rows of different datasets must satisfy to be combined (e.g., they share the same standardized product identifier). The join operator can also be equipped with some sophisticated techniques for joining rows, such as one based on entity resolution.

Data expansion. It is the process of putting together rows of two datasets that contain data referring to different observations of the same phenomenon or to different objects of the same type. This can be useful when, for instance, a training set is built by accumulating data coming from diverse data sources, say D_1 , D_2 and D_3 (e.g., experimental data of a medical treatment produced by three different laboratories). The append operator \uplus can be used in such scenarios possibly preceded by some data reduction operators, for example as follows:

$$D_1 \uplus \pi_{C_1}(D_2) \uplus \sigma_{C_2}(D_3)$$

. As shown in Example 2.5, this operator also accounts for situations in which we need to merge datasets that involve different features of the same phenomena.

4 Abstract analysis of provenance capture

In order to capture the provenance of a pipeline p of a combination of pre-processing operations $\mathbf{o}_1, \dots, \mathbf{o}_n$ forming a tree, we introduce an abstract provenance-generating function (*prov-gen*), and associate it with each operation \mathbf{o}_k occurring in p .

In accordance with the provenance model presented in Section 2.3, each element d_{ij} of a dataset D produced during the execution of p is represented by a PROV entity in the provenance document. The properties of this entity include the row index i and feature j in D , and an identifier k denoting the fact that d_{ij} is in the result of the operation \mathbf{o}_k in p .

Similarly, each operation \mathbf{o}_k in p is represented by a PROV *activity* in the provenance document, whose properties specify the operator(s) illustrated in Section 2.2 that implement(s) \mathbf{o}_k , and the list of the features on which \mathbf{o}_k operates.

4.1 Provenance templates

We now present example instances of provenance-generating (*prov-gen*) functions for the main types of operations observed in data science pipelines, discussed in Section 3. To recall, these are: (i) data reduction: $D' = \pi_C(D)$, $D' = \sigma_C(D)$; (ii) Data augmentations: $\alpha_{f(X):Y}^{\rightarrow}$, $\alpha_{X:f(Y)}^{\downarrow}$; (iii) Data transformations: $\tau_{f(X)}$; and (iv) Data fusion.

A prov-gen function takes as inputs the sets of input and output values D, D' for the operator, and produces a PROV document that describes the transformation produced by the operator on each element of D , as reflected in D' . Note that for binary operators, namely join and append, D includes inputs from both operands.

Take for example the case of Vertical Augmentation (VA): $\alpha_{f_1(\mathbf{Age}):ageRange}^{\rightarrow}(D)$ which we used in Example 2.3, where attribute **Age** is binarised into $\{young, adult\}$ based on a pre-defined cutoff, defined as part of $f()$. The prov-gen function for VA will have to produce a collection of small PROV documents, one for each input-output pair $\langle D_{i, \mathbf{Age}}, D'_{i, \mathbf{AgeRange}} \rangle$ as shown in the example.

As these documents all share the same structure, we define a common *PROV template* which is then instantiated multiple times, once for each input/output pair. A template is simply a PROV document that may contain variables, indicated by the namespace **var:**, which are used as placeholders for values. Here templates are designed to capture the transformation at the level of individual elements of D , or its rows or columns, as appropriate. Thus a template will have a *used* set of entities, which refer to the subset of data items in D which have been used by \mathbf{o} , and a *generated* set of new entities, corresponding to new elements in D' (for projection and selection, it will have an *invalidated* set of entities instead, as these operators remove data from D).

The PROV template for (VA) is shown in Figure 3, where we use the generic attribute names X, Y to indicate the old and new feature names. One or more *binding generators* are associated with each template: they determine how values found in D, D' upon execution of the operator are substituted for the variables. Each variable substitution results in a small PROV document, which represents all derivations through a single operator. In the following we are going to refer informally to these documents as *provlets*, to indicate that a complete PROV document representing a complex derivation chain can be produced by joining multiple such provlets on their data identifiers, as described in Sec 6.3 below.

In the VA example, the transformation between D and D' is 1:1 and thus a new provlet is created from each value $D_{*, \mathbf{Age}}$ of column **Age** and the corresponding value in **AgeRange**.

Using a list comprehension notation, the binding generator for the variables used in the template in Figure 3 are defined as:

$$[\langle F = \mathbf{Age}, I = i, V = D_{i, \mathbf{Age}}, F' = \mathbf{AgeRange}, J = i, V' = f(D_{i, \mathbf{Age}}) \rangle | i : 1 \dots n]$$

These are the new entities for the newly created data elements in the new column $D_{*, \mathbf{AgeRange}} \in \{young, adult\}$. Two of the n PROV documents for this specific example are shown in Figure 3.

4.2 Template binding rules

We define templates for each of the five core operators, shown in Figure 4 and the corresponding binding generators for *used*, *generated*, and *invalidated* sets of entities.

Note that we do not need to create complete provlets for all entities in any given output dataset. If $f(D)$ does not change d_{ij} , then no provenance record needs to be generated. However if $f(D)$ discards elements of D , then a provlet containing an invalidation relationship is required. Whenever a new entity is generated, i.e. when $f(D)$ creates a new

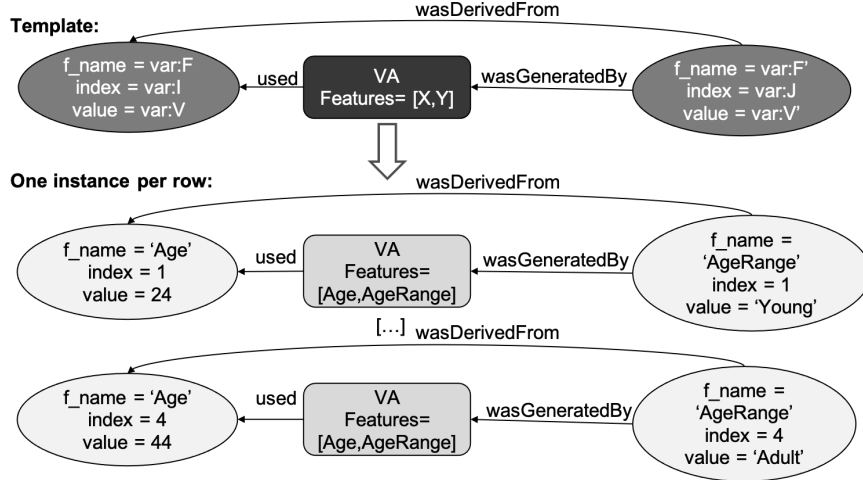


Figure 3: Example of PROV template for Vertical Augmentation and corresponding instances.

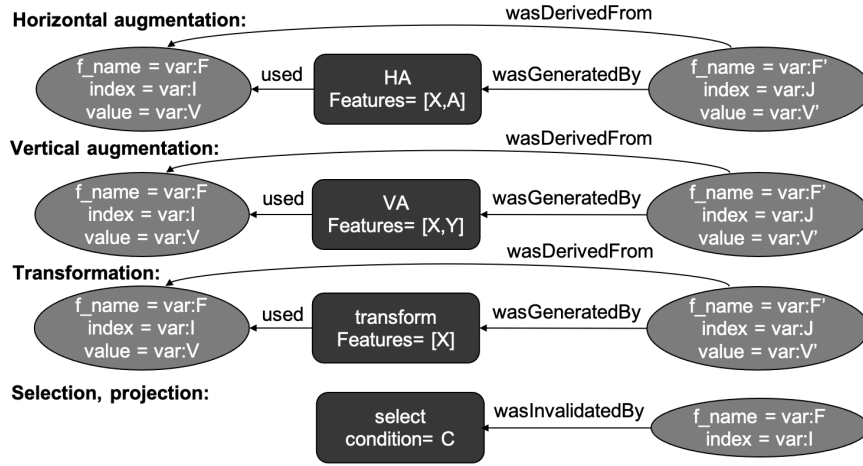


Figure 4: PROV templates used by the prov-gen functions for data augmentation, transformation, and reduction.

or updated value in d_{ij} , a complete provlet is also required. In other words, we only require provenance statements that capture different versions between elements in the dataset.

4.2.1 Data reduction, selection

Data reduction *invalidates* existing entities. For selection: $D' = \sigma_C(D)$, the bindings specify that an entire row i is invalidated whenever condition C is False when evaluated on that row. This affects all features $X \in S$:

$$[\langle F = X, I = i \rangle | X \in S, i : 1 \dots n, C(D_{i,*} = \text{False})]$$

A *wasInvalidatedBy* relationship is established between each of these entities and a single *Activity*, representing the selection.

4.2.2 Data reduction, projection

Conditional projection $D' = \pi_C(D)$ invalidates all elements in column $X \in S$ whenever C returns True when evaluated on elements of X :

$$[\langle F = X, I = i \rangle | X \in S, i : 1 \dots n, C(D_{*,X} = \text{True})]$$

Similar to the selection, here too a *wasInvalidatedBy* relationship is established between each of these entities and a single *Activity*, representing the projection.

4.2.3 Vertical augmentation

$\alpha_{f(X):Y}^{\rightarrow}$ takes a set $X \subset S$ of features and adds a new set Y of features, $Y \cap S = \emptyset$ to D' as shown in Ex. 2.3. The provenance consists of n PROV documents, one for each row i of D , and in each such document entities for $D_{i,X_m}, X_m \in X$ are used to generate entities for the new features $Y_h \in Y$. Thus, the bindings are defined as follows:

For $i : 1 \dots n$:

used entity: $[\langle F = X_m, I = i, V = D_{i,X_m} \rangle | X_m \in X]$

generated entity: $[\langle F' = Y_h, J = i, v = f(D_{i,X}) \rangle | Y_h \in Y]$

These entities are then connected to a single *Activity*, as shown in Figure 4 and in the examples (Fig. 3, 14), using *Used* and *wasGeneratedBy* relationship. For each pair of *used*, *generated* entities having the same index on each side (i.e., where **var:I = var:J** after template instantiation), a *wasDerivedFrom* relationship is also added, to assert a stronger relationship (derivation occurs through the Activity that connects the entities).

4.2.4 Horizontal augmentation, grouping and aggregation

The $\alpha_{X:f(Y)}^{\downarrow}$ operator groups records according to columns $X \subset S$, producing a list $G = [g_1 \dots g_h]$ of h groups. Then for each $g_i \in G$ it computes $f(Y)$ from the records in the group, producing a new record containing the aggregated value in column A , the values that define the group in each column $X_m \in X$, which we denote $val(X_m, g_i)$, and *null* in all other columns (see Ex. 2.3 in Section 2.2). Thus, the operator produces h records, and let $rows(G) = [n + 1, n + 2, \dots, n + h]$ denote their new row indexes in the dataset.

The corresponding provenance template and binding rules are similar to those for Vertical Augmentation (Figure 4), but with some differences, and are best illustrated initially using an example. Consider the following dataframe:

	X₁	A	B
1	x_1	10	b_1
2	x_2	30	b_2
3	x_1	20	b_3
4	x_2	40	b_4
5	x_1	30	\perp
6	x_2	70	\perp

and grouping operator $\alpha_{X:f(Y)}^{\downarrow}$ where $X = [X_1]$, $f(Y) = \sum A$, where the sum on A values occurs for each group.

The set of provlets that represent the derivations of the elements in the new rows 5, 6 are depicted in Fig. 5. Values x_1, x_2 in rows 5,6 identify the groups and are derived from the corresponding values in rows 1,3 and 2,4, respectively. Similarly, the two values in column A are obtained by adding up the corresponding A values in the same groups of rows (1,3 and 2,4). Finally, the null values in column B are generated by the operator, but their values are not derived from any inputs.

Generating these provlets requires maintaining the association between each group: group 1 in row 5, and group 2 in row 6, and the corresponding input rows (1,3 and 2,4). Implementations can achieve this in different ways. Formally, we assume that each group $g_i \in G$ maps to a set of “group input” rows, i.e., $ginput(g_i)$. In our example, we have $ginputs(g_1) = \{1, 3\}$, $ginput(g_2) = \{2, 4\}$.

Then, the binding rule for group g_i in row i and elements in each of the columns $C \in X$ can be written as follows. For generated entities:

$$\langle F = C, I = i, V = D_{i,C} \rangle$$

For used entities:

$$[\langle F = C, I = j, V = D_{i,C} \rangle | j \in ginput(g_i)]$$

Similarly, for the values in columns $C' \in Y$, the rule for generated entities is:

$$\langle F = C', I = i, V = D_{i,C'} \rangle$$

and for used entities:

$$[\langle F = C', I = j, V = D_{i,C'} \rangle | j \in ginput(g_i)]$$

Finally, the rule for the null values applies to values in columns $C'' = S \setminus Y \setminus X$, and they only have the *generatedBy* side:

$$\langle F = C, I = i, V = Null \rangle$$

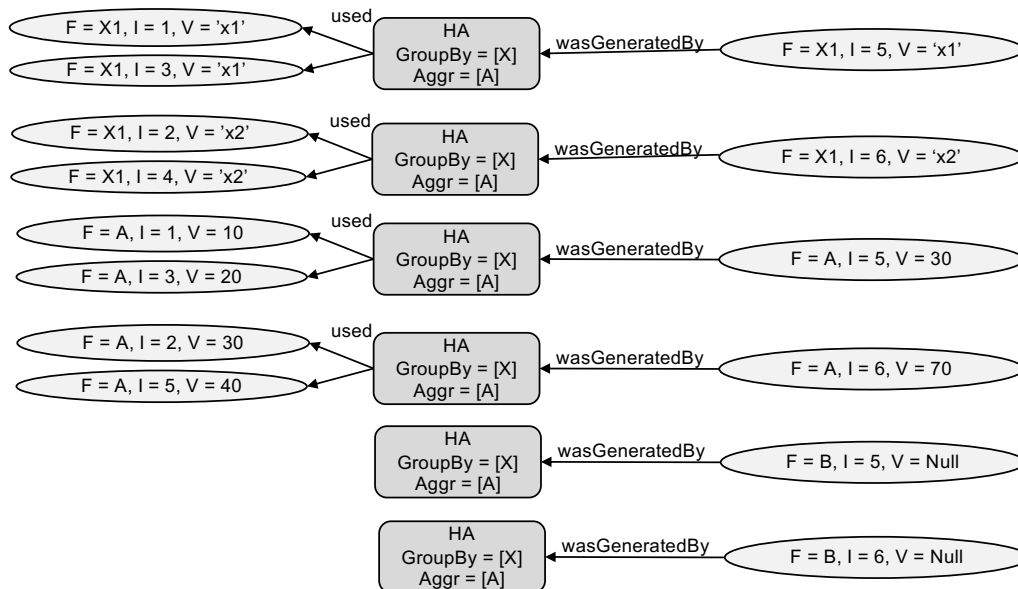


Figure 5: Provlets describing grouping and aggregation from Example in Sec.4.2.4

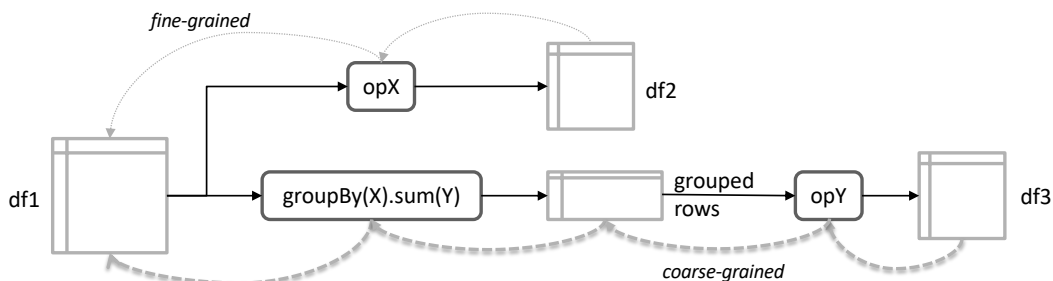


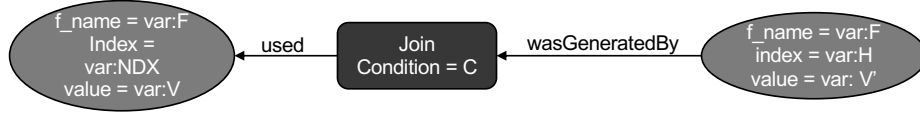
Figure 6: Aggregation pattern and loss of provenance granularity

Note that aggregation operators reduce the granularity of the derivations. Typical Value Transformation operators, for instance a normaliser, would map each input element to a corresponding output element. Aggregations, on the other hand, produce a “provenance bottleneck” where n rows are mapped to $m < n$ rows, where m is the number of groups, because the provenance of any “downstream” dataframe that makes use of the groups will have to include one of the group rows. In practice, aggregations may produce a pipeline pattern as shown in Fig. 6, where some of the operators (opY) use the aggregations, and the provenance of new dataframe elements produced by these operators will map to grouped rows and not to the upstream un-aggregated dataframes, leading to some loss of granularity. In the Figure, the bottom provenance dependencies (thick dotted lines) for elements of $df3$ must include some of the group rows, and those in turn are derived from *each* of the inputs. Note also, however, that the loss of granularity depends on the number of groups. In the extreme case where the grouping operator produces a single group consisting of all input rows, for instance, the result is a provenance graph where all inputs contribute to the grouping, and all outputs depend on the grouping, producing a complete bottleneck.

4.2.5 Data transformation

$\tau_{f(X)}$ takes features $X \subset S$ and computes derived values, which are used to update elements of D , but without generating new elements. The bindings reflect such in-place update, but as the new value for each element is defined by $f()$, we assume for simplicity that *all* values are updated, although, in reality, some will stay the same, as shown for instance in Ex. 2.4 (imputation). The resulting bindings reflect this many-many relationship, where (potentially) all values in a column $X_m \in X$ are used to update (potentially) all values in that same column (and this applies to

Pattern 1: applies to all triples $\langle D_{i,f}^L, D_{j,f}^R, D'_{h,f} \rangle$ where $f \in F$



Pattern 2: applies to pairs:

$\langle D_{i,f}^L, D'_{h,f} \rangle$ where $f \in S^L$, $\langle D_{j,f}^R, D'_{h,f} \rangle$ where $f \in S^R$

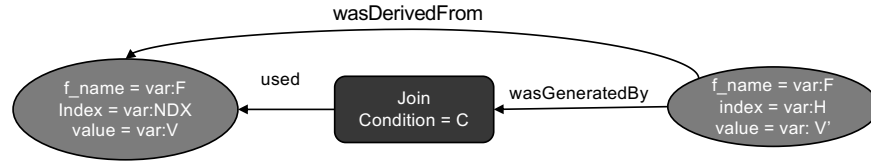


Figure 7: PROV templates for joins.

each column). Thus, the provenance document consists of $|X|$ provlets, one for each column, with bindings defined as follows. *Used* entities:

$$[\langle F = X_m, V = D_{i,X_m}, I = i \rangle | i : 1 \dots n]$$

Generated entities:

$$[\langle F' = X_m, V' = f(D_{*,X_m}), J = i \rangle | i : 1 \dots n]$$

Used and *wasGeneratedBy* relationships, mediated by an *Activity*, are created between each *Generated* entity and all of the *Used* entities having the same X_m , along with the corresponding *wasDerivedFrom* relationships.

It is worth clarifying one potential limitation that occurs when the data derivation operator contains parameters whose values are set by inspecting the input dataframe. In our approach, these values are not “used” by the operator, despite the fact that, in reality, the operator is input-dependent. As an example, consider a Scaling operator, which scales each value in column X using a range that is defined by the min and max values found in X . According to the template just defined, this operator produces a set of 1-1 derivations, namely from each output value in X , back to its corresponding input value. However, in the current approach the fact that the Scaler depends on the input values of X , which it has inspected, is not captured.

4.2.6 Join

In Section 2.1 we introduced a join operator: $D' = D^L \bowtie_C^t D^R$ where condition C may involve any columns $F \subset S^L \cup S^R$. As an example, consider $S^L = [A, B, C]$, $S^R = [A, C, D, E]$ and $C \equiv D^L.A = D^R.A$ **and** $D^L.B = D^R.D$, thus $F = \{D^L.A, D^R.A, D^L.B, D^R.D\}$.

Let $D_i^L = [x, y, c_1]$, $D_j^R = [x, c_2, y, e]$ be two tuples that contribute a result tuple $D'_h = D_i^L \bowtie_C^t D_j^R = [x, y, c_1, x, c_2, y, e]$.

Note that D_i^L, D_j^R correspond precisely to the *witness* tuples in the *why-provenance* of $D'_{h,f}$, for some attribute $f \in F$, as defined in [22]. The *why-provenance* of $D'_{h,f}$ can be expressed formally in terms of the two contributing tuples, i.e., using the polynomial notation proposed in [25]. However, here we are interested in the more granular derivations at the level of the single values, rather than of the entire tuple. To express the fine-grained provenance of a value $D'_{h,f}$ in the result, we first consider the values $D_{i,f}^L, D_{j,f}^R, f \in F$, *used* by the join operator to evaluate C :

$$used = \{D_{i,f}^L \cup D_{j,f}^R | f \in F\} = [D_{i,A}^L = x, D_{i,B}^L = y, D_{j,A}^R = x, D_{j,D}^R = y]$$

We apply template (1) in Fig. 7 to assert that each value in $D'_{h,f}$ *was generated* by the join operator and that the operator *used* all the values in the *used* set.

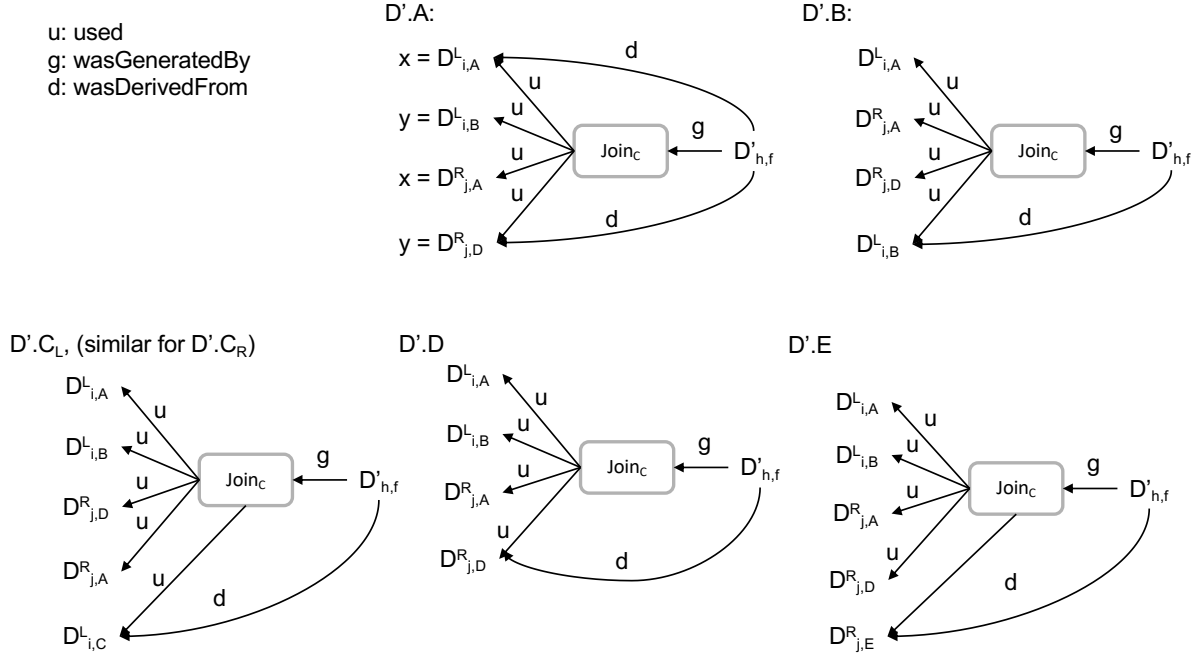


Figure 8: Instantiated PROV templates for the example in the text

This is achieved using the following binding generator:

$$\begin{aligned}
 &\text{for } f \in F : \\
 &\quad \text{if } f \in S^L : \langle F = f, NDX = i, H = h, V = D^L_{i,f}, V' = D'_{h,f} \rangle \\
 &\quad \text{if } f \in S^R : \langle F = f, NDX = j, H = h, V = D^R_{j,f}, V' = D'_{h,f} \rangle
 \end{aligned}$$

Secondly, we express that each value in the result was *derived from* the corresponding value in one of the two operands, and that the derivation is supported by a *usage/generation* pair as shown in template (2) in Fig. 7. Note that this template covers both the case where a feature is used as part of an equijoin condition, such as *A* in the example, and also the case where null values are generated as part of an outer join. Template 2 is instantiated using the following bindings generator:

$$\begin{aligned}
 &\text{for } f \in S^L : \langle F = f, NDX = i, H = h, V = D^L_{i,f}, V' = D'_{h,f} \rangle \\
 &\text{for } f \in S^R : \langle F = f, NDX = j, H = h, V = D^R_{j,f}, V' = D'_{h,f} \rangle
 \end{aligned}$$

Each of the two templates generates a PROV fragment, and these are then combined by virtue of their common entities and activity (the join operator). Fig. 8 (where the actual values of $D'_{h,f}$ are only shown in the first provlet, to avoid overloading the Figure) shows the provenance fragments for values $D'_{h,f}$ for a generic tuple h and for each f . Note in particular that the generation relationships in templates 1 and 2 do not result in multiple generation arcs in the final provenance, as those have identical source and sink nodes (i.e. the entity representing the value and the activity representing the join).

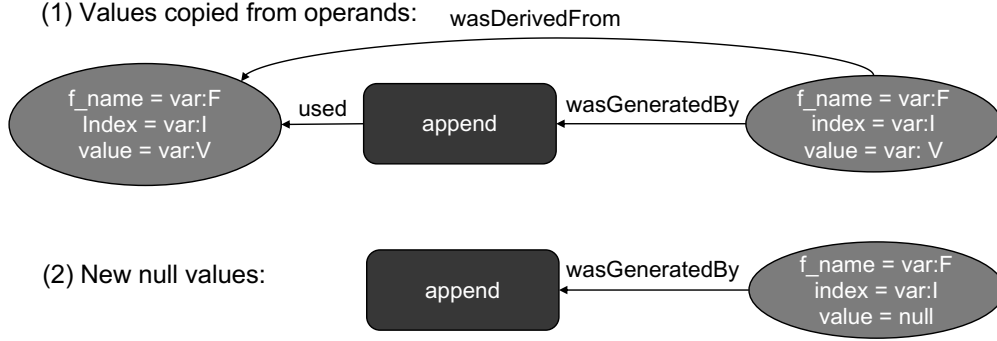


Figure 9: PROV template for append

4.2.7 Append

Consider again Example 2.5 in Section 2.1 (page 3), where a dataset D^L with schema $S^L = [\text{CId}, \text{Name}]$ is appended to D^R with schema $S^R = [\text{CId}, \text{Gender}, \text{Age}, \text{Zip}, \text{Name}]$: $D' = D^L \uplus D^R$. Let n, m be the number of rows in D^L, D^R , respectively. Observing that the order of the rows in the operands is preserved in the result, we identify four types of output values $D'_{i,f}$: (1) values derived from a corresponding $D^L_{i,f}$, when $i < n_1$ and $f \in S^L$; (2) values derived from a corresponding $D^R_{i,f}$, when $i \geq n_1$ and $f \in S^R$; (3) Null values when $i < n_1$ and $f \notin S^L$; (4) Null values when $i \geq n_1$ and $f \notin S^R$.

A derivation relationship is created for cases (1) and (2), which is supported by a corresponding generation-usage pair of relationships, with the operator as the mediating activity; while for cases (3) and (4), only a generation relationship is created. Fig. 9 shows the PROV template for this pattern.

The binding generator function for derivations, generation, and usage of copied values is defined as follows:

for $i : 0 \dots n_1 - 1$: if $f \in S^L$ then $\langle F = f, NDX = i, V = D'_{i,f} \rangle$ # template (1) applies
 for $i : n_1 \dots n_2 - 1$: if $f \in S^R$ then $\langle F = f, NDX = i \rangle$ # template (2) applies

5 Provenance generation

The combination of provenance templates and corresponding binding rules, embodied by the *prov-gen* functions, which we just presented, provide a formal description of the *provenance semantics* associated with each of the core operator classes: data reduction, augmentation, transformation, and fusion. In this section, we present a concrete approach to provenance generation that is grounded in this formalisation.

5.1 The approach

Provenance generation operates by (i) observing the execution of operators that consume and generate datasets, (ii) analysing the value and structural changes between the input(s) and output datasets for that command, and (iii) based on the observed change pattern, select one or more of the templates described in the previous section, to capture the dependencies between the elements of the datasets that have changed. This approach ensures that the topology of the resulting provenance graph is consistent with the templates, but it also broadens the scope of the operators for which provenance is generated, namely to *any operator* that transforms an input into an output dataset.

As a simple example, consider an imputation operation that causes some previously null values to be set to 0, in some (or all) of the columns. This values change pattern is easily recognised and is used to trigger provlet generation using the appropriate template, in this case *data transformation* (cf. 4.2.5).

More general transformation patterns can be captured using more than one template, and by composing the resulting provlets. For example, consider a pipeline like the following, in which D_a, D_b , and D_c are the input datasets and f is

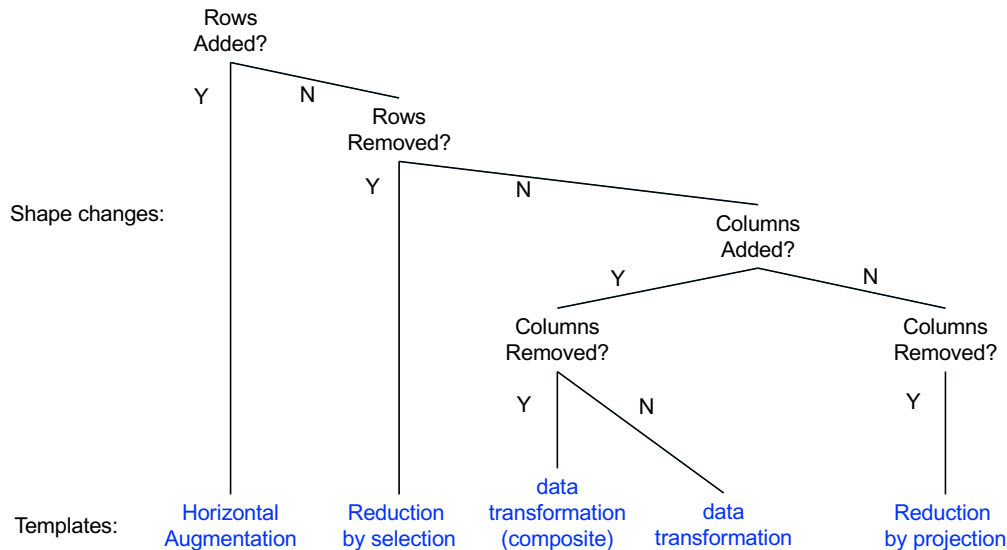


Figure 10: Template selection for shape change.

an imputation function over a feature K of D_a :

$$\begin{aligned}
 D_1 &= \tau_{f(K)}(D_a) \\
 D_2 &= D_b \bowtie_{K_1=K_2}^{\text{outer}} D_c \\
 D_3 &= D_1 \uplus D_2
 \end{aligned}$$

Its execution results in a collection of three provlets, each accounting for the dependencies between elements of the datasets (1) D_1 and D_a ; (2) D_2 and D_b, D_c ; and (3) D_3 and D_1, D_2 , respectively. At the end of the execution, these provlets are consolidated into a single, final provenance document that accounts for all transformations across the entire pipeline. In this example, this will create a graph of dependencies where elements of D_3 are linked through derivation relationships to elements of D_a, D_b, D_c .

In the cases above, the change analysis identifies the appropriate template without the need for syntactic analysis of the source code. In particular, these examples illustrate *simple provenance generation*, so called because provlets are independently generated for each input/output datasets pair. More complex *composite provenance generation* can also be achieved, which captures the provenance of an operation implemented by a sequence of commands. We illustrate this in the next Section for the case of one-hot encoding transformation.

5.2 Change analysis algorithm

We now present the dataset change analysis algorithm that is responsible for generating each of the provlets. The algorithm considers unary and binary operators separately, with help from lightweight code instrumentation. In the following, we only discuss the case of unary operators, as a complete example of join and append provenance has been provided earlier. Implementing join provenance efficiently presents new challenges, however, and these are discussed separately below (Section 6.4). Details of the code instrumentation required to support provenance generation are provided in the next Section, along with details of the *Observer* pattern [26] used to monitor changes in datasets through execution. The algorithm looks at changes in either shape or values between the input and output datasets, denoted D and D' , respectively. The cases listed below are summarised in Figures 10 and 11. Shape changes are detected simply by comparing the number of rows m, m' or the number of columns n, n' in D, D' . Value changes are detected by reviewing values within each column.

Shape changes. When $m' < m$ or $m' > m$, the horizontal augmentation (cf. 4.2.4) or reduction by selection (cf. 4.2.1) templates are applied, respectively. Adding columns is interpreted similarly, i.e., $n' > n$ triggers the application of the Vertical Augmentation template (4.2.3). However, this condition also causes a list to be created which contains the added columns. If D' is then used as input to the next command, producing D'' , and it is the case that $n'' < n'$ when the list of added columns is non-empty, then this is interpreted as a sequence where first a set of columns are added, and then other columns are removed. This enables the provenance generator to infer dependencies between

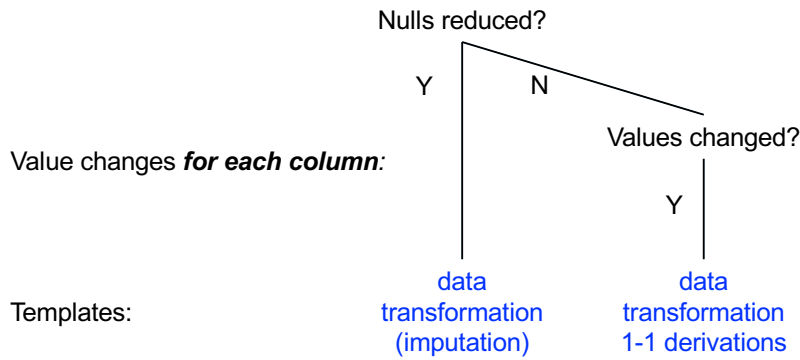


Figure 11: Template selection for value change.

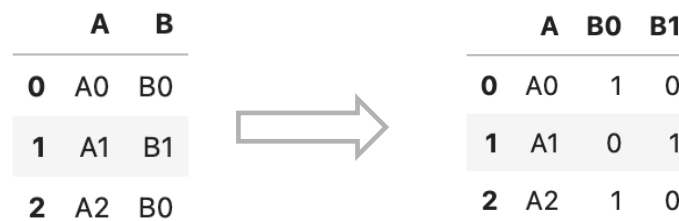


Figure 12: One-hot encoding transformation

such columns. Derivation relationships are thus added accordingly to the provlet that represents the provenance for D , D' , and D'' .

This composite behavior makes it possible to detect patterns like one-hot encoding, which both adds and removes columns but does so using more than one operator as shown in Figure 12.

The following sequence of operations, in which an input dataset D is first extended by encoding with the function h the values occurring in the feature B and then the such feature is deleted, is routinely used to achieve the result.

Example 5.1 (one-hot encoding)

$$D_1 = \alpha_{h(B)}^{\rightarrow}(D)$$

$$D_2 = \pi_{\{A \text{ and the features not occurring in } D\}}(D_1)$$

After the first operation, the generator would only know that a number of columns have been added (one for each value in column B) but would be unable to determine any other dependencies. A list is created with these column names, which will only exist within the scope of the next command. Executing the second operation results in column B being removed. Rather than two connected provlets, here a single provlet is generated, which accounts for the change in dataset structure, and where derivation relationships are added between each new column and column B (which is then itself invalidated in the provlet). In Fig. 10 we refer to this as the *composite data transformation* template.

Example 5.2 (Provenance of one-hot encoding) Consider the transformation in Fig. 12 implemented by the operations in Ex. 5.1. After execution of the first operation, a *VerticalAugmentation* activity is created to account for the generation of the new features. However at this stage, we do not know which elements of the input have been used, thus we are also unable to add derivation relationships. After executing the second operation, feature B has been removed, and in the provlet this is recorded by introducing a new *ConditionalProjection* activity that invalidates B . Additionally, however, the composite variant of data transformation mentioned above is applied, resulting in a new relationship:

VerticalAugmentation used B

as well as derivations:

B0 wasDerivedFrom B,

B1 wasDerivedFrom B.

The complete provlet includes the statements:
B wasInvalidatedBy ConditionalProjection
B0 wasDerivedFrom B
B1 wasDerivedFrom B

Value changes. This analysis considers one column at a time. If some or all of the values have changed in a column C , the *data transformation* template is applied, with the assumption that there is a one-to-one dependency between each new value d'_{ij} of D' and the corresponding original value d_{ij} of D , and this is mediated by the function represented by the operator, as described in Section 4.2.5. The case of value imputation is handled separately. This is detected simply by comparing the number of null values in D (identified by NaN) to those in D' . Imputation has occurred when the nulls have been reduced. In this case, the *data transformation* template is used (4.2.5), where by default each value in an imputed column C in D' are assumed to be derived from all values in the same column C in D . Notice that the generator does not have further information to make the derivation more granular. Also, when multiple columns have been imputed, each of those is considered independently from the others. This may miss derivations, again for lack of information. For instance, using the MICE algorithm [27] will impute multiple columns, where each new value is derived from values in multiple source columns. This generalisation is not captured by the algorithm.

5.3 Benefits and limitations of the change analysis approach

The approach of using dataset change as the trigger to choose the provenance template and to apply and generate provenance information has two main advantages. Firstly, it makes it possible to capture provenance when the internal logic of the operators is not accessible to the observer. This has been referred to as the “black-box” problem by the provenance community [28]. Secondly, it enables capturing the provenance of operator compositions. In the presentation of this work, we mainly describe the provenance generated for a single operator execution. However, by looking only at dataset change, we can allow multiple operators to execute and generate the provenance record for this group of operators. An example is the “stateful” shape change analysis above, which keeps track of data transformations across more than one operator, in order to accurately infer derivation dependencies.

One limitation that is intrinsic to this approach is in complex cases such as when UDFs are employed. In this case, while the algorithm can detect which tuples have changed, it cannot identify which inputs caused the change, thus it must assume that all inputs were used by default.

The ability to group operators is beneficial for many reasons. Provenance is often unwieldy, capturing interactions and relationships meaningless for later use. Past works utilize variations in “Composite” to help with various tasks. ZOOM [29] used the concept of composite step-classes to develop a notion of user views, allowing a user to more easily view and understand a provenance graph. More recently, Ursprung [30] contains provenance at different composite levels based on the capture mechanism able to be deployed in a given situation. In our work, the developer can choose a composite that is correct for their ultimate end needs by having the provenance observer wait for other commands to complete and only look at the final dataset.

6 Implementation and Architecture

In this section, we provide details on (i) the data architecture used in the implementation, (ii) the code instrumentation required for the provenance generator to operate, and (iii) the efficient implementation of provenance capture and provlet composition.

6.1 System architecture

We have created a reference implementation of the approach to provenance generation illustrated in Section 5 using the pandas/python library, representing datasets as pandas dataframes². The overall architecture for provenance capture, storage, query, and visualisation is shown in Fig. 13. The Provenance-Tracker automates the process of detecting and tracking the provenance of a user-defined pipeline of data preparation. It includes a Prov-generator that produces the provenance of each operator in the pipeline by analyzing its effect on the underlying dataset. This is done at execution time by: (i) identifying the operator under execution on the basis of a series of comparisons between the input and the output datasets, (ii) executing the prov-gen function of the core operation that captures the identified operator by suitably instantiating the function template, and (iii) storing the provenance data produced by the prov-gen function on an underlying repository. Since provenance data have a natural graphical representation, Neo4j, a world-leading,

²<https://pandas.pydata.org/>

industry-grade, scalable graph database management system, is used for this purpose. Note that while provenance graphs are written to the database at runtime, i.e., while the script is executing, those writes can happen asynchronously, as the graph will only be queried “post mortem” after the script has finished executing. This also removes the need to consider a high-performance back end such as an in-memory database.

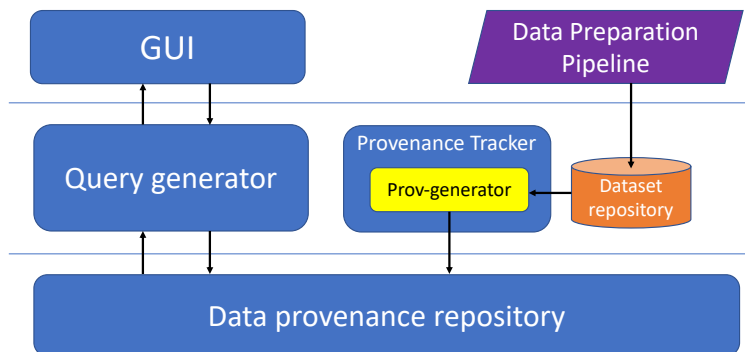


Figure 13: System architecture

The Query-generator allows the user to perform several types of analyses of the data provenance collected for a given data preparation pipeline, by translating a specific data-provenance exploration chosen from a menu of a graphical interface into a query expressed in Cypher, the query language of Neo4j, as it will be illustrated in Section 7.4.

With the current reference implementation we have made several upgrades to the earlier version [31]: (1) the data representation format; (2) the storage method, as we now use the Neo4J graph database to store the final provenance graph natively in contrast to [31], where all provenance was serialized in PROV-JSON [9, 20] (an interoperability format for the PROV data model); and (3) *observing* provenance from dataframes instead of specifically coding each pandas operator. We have chosen to represent provenance graph using the standard PROV data model, to ensure some degree of interoperability across applications that want to use the provenance graphs. However, we are also aware that the standard PROV serialisations documented as part of the W3C specification are not concerned with space utilisation and query performance. Thus, our prototype implementation aims to strike a balance between performance and interoperability goals. In particular, all framework elements that have changed through an operator are materialised in the provenance (but no new entities are introduced to represent data items that have not changed). As all required entities are manifested in the graph, new provenance queries can be written simply using standard Cypher, with minimal knowledge of the internal representation. In a future implementation, one may introduce entities that represent entire tuples or columns, but with the understanding that queries must be aware of these optimisations.

6.2 Code instrumentation

A number of different approaches for capturing provenance from a running process have been documented in the literature. These range from intentionally placing capture calls within the notebook, to utilizing libraries to compare dataframes for automatic detection, to engaging interactively with the user. A key distinction concerns how much burden can be placed upon the user. Works such as [32] or [33] insist on no-human involvement, while others believe that users should be invested in the process of improving their scripts, specifically [34] or [35] allow users to enter provenance capture calls at appropriate places. Other contemporary systems, such as MLInspect [36], require the development of specialised add-ons to the code (using a visitor pattern) to create observers. The level of developer involvement is still an open research question for the data science community.

In this work we aim to implement the strategy described in Section 5.2 while minimising user intervention. This is achieved using an Observer software pattern that acts as a wrapper for dataframes and relies on a Provenance Tracker object for deriving the provenance of a transformation based on the inspection of the dataframes in input and output. From an operational standpoint, the Provenance Tracker is equipped with a `subscribe()` function that allows users to subscribe to one or multiple dataframes for tracking their provenance. Basically, the invocation of this function returns the corresponding wrapped dataframes as objects that encapsulate nearly all of the methods inherited from the pandas DataFrame class, enabling provenance generation in a transparent way during dataframe transformations. The only required instrumentation is the following:

```

tracker = ProvenanceTracker()
df, df2 = tracker.subscribe([df, df2])

```

After this, the signature and syntax of methods that operate on a dataframe remain unchanged, as in the examples that follow. However, they now operate on the wrapped dataframes and invoke the internal provenance-capture functionality through the Provenance Tracker.

```

# Imputation
df = df.fillna('Imputation')
# Feature transformation of column D
df['D'] = df['D'].apply(lambda x: x * 2)

```

Similarly, provenance generation for the join operation can be done without the need to invoke additional auxiliary functions, as follows.

```

df = df.merge(right=df2, on=['key1', 'key2'], how='left')

```

The activity of the Provenance Tracker can be temporarily disabled to capture the provenance of an operation made of several basic data transformations. This is done by using the `dataframe_tracking` property as in the example that follows, which implements the provenance capture of the one-hot encoding sequence illustrated in Example 5.1.

```

tracker.dataframe_tracking = false
dummies = pd.get_dummies(df['B'])
df = df.concat(dummies.add_prefix('B'+'_'))
tracker.dataframe_tracking = true
df = df.drop([c], axis=1)

```

In this example, the changes made by the horizontal augmentation on the original dataframe are produced but taken into account only during the subsequent operation, when the `dataframe_tracking` property is set to 'true'.

6.3 Composing provlets into a complete provenance document

A complete provenance document is produced by combining the collection of provlets that results from each instance of change analysis. Specifically, one provlet is generated for every transformation and every element in the dataframe that is affected by that transformation. The final document is composed of such a collection of provlets, where entity identifiers match across provlets, and never needs to be fully materialised, as explained shortly.

Consider for instance the following pipeline:

$$\sigma_C(\alpha_{\vec{f}_1(\text{Age}): \text{ageRange}}(D))$$

where $C = \{\text{AgeRange} \neq \text{'Young'}\}$ and D is the dataset of Example 2.3. The corresponding provenance document is represented in Figure 14. Applying vertical augmentation produces one provlet for each record in the input dataframe, showing the derivation from **Age** to **AgeRange**. The second step, selecting records for 'not young' people, produces the new set of provlets on the right, to indicate invalidation of the first record, as per the template at the bottom of Figure 4. Note that the "used" side on the left refers to existing entities, which are created either in the pipeline from the input dataset, or by an upstream data generation operator.

Provlet composition requires looking up the set of entities already produced, whenever a new provlet is added to the document. One simple way to accomplish this is by eagerly keeping the entire document in memory, along with an index for all entities, and by mapping each entity to the corresponding data element it represents. While this can be accomplished using readily available Python PROV libraries [37], it does not scale well to the volume of entities required to represent large dataframes in cases where more than a handful of transformation operators are involved. Instead, we have followed a continual append approach for provenance composition in which each p-gen function

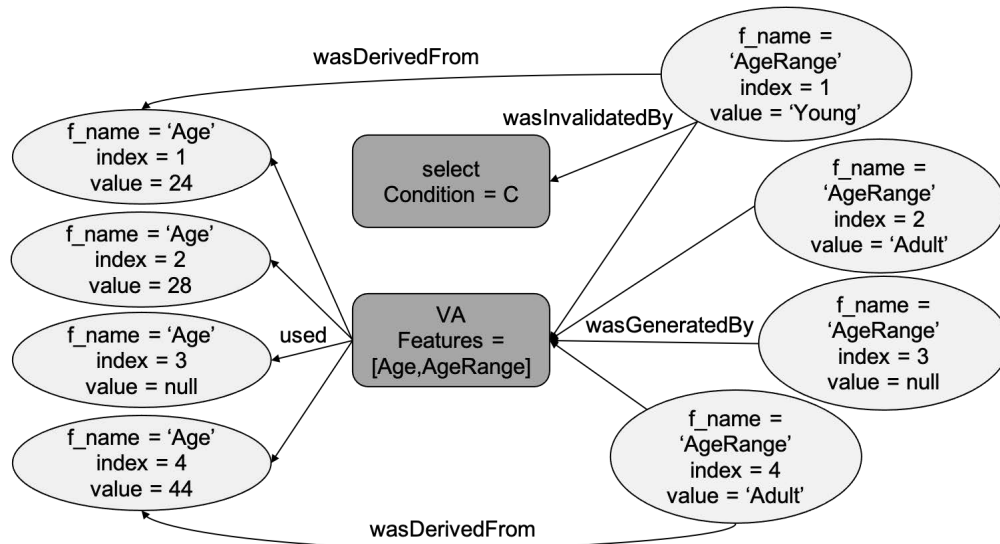


Figure 14: Provlet composition.

generates a set of provlets (in the worst case one for each element in the dataframe) that are just collected in a partial document and stored in the underlying repository. This allows the provenance to be collected quickly at the execution of each script, and be assembled later, minimizing execution dependencies and possible bottlenecks during the actual execution of the pipeline.

6.4 Efficient provenance generation

The overhead for provenance collection and composition described above can be minimised by using Python’s multiprocessing library to parallelise the most expensive operations, observing that (i) dataframes can be split into chunks and provenance entities generated independently for each chunk, and (ii) the provlets generated by each parallel process can be independently written to disk, and then asynchronously inserted into Neo4J. Assuming that provenance graphs are only queried after the end of script execution, this provides a scalable back-end solution despite the potential limitations of Neo4J’s centralised architecture.

Using parallel processes to write provlets to disk is straightforward, as there are no dependencies amongst these processes. As an example, for one-hot encoding provenance consisting of about 2M entities, we observe a stable 70% improvement in writing times using 12 processes, relative to a sequential baseline. In practice, at most one chunk is created for each available CPU thread and allocated to one process. One slight complication is that assigning each generated entity to its corresponding dataframe element requires keeping track of the relative order of the chunks in the dataframe. This is accomplished using a queue (further details omitted). Unlike for write operations, here performance gains depend on the complexity of the specific operator, i.e., of the template used. Empirical results indicate an average of 60% improvement relative to the sequential baseline. Performance figures from our comprehensive evaluation are reported in the next section.

Joins present an interesting implementation twist to provenance generation mechanism. A naive implementation of join provenance that creates instances similar to template in Section 4.2.6, would simply link each row of the output dataframe to the two input DataFrames using rules to infer the derivations for every single item in a row. Unfortunately, joins expose one of the problems of our approach which looks at the input/output datasets and not the operator itself. Because we are not linked directly to the join operator, which may or may not have the standard guarantees of a database system, re-creating which rows in the input dataframes and their relationship to the output row in the dataframe takes effort.

Consider the naive implementation of creating join provenance records using our data-observation approach. For every row in the output dataframe, the join key(s) must be identified within the data, and the actual data values in the remaining features noted. Then, the input dataframes must be scanned to locate the key(s), and the row examined to determine if it contains the appropriate data values to match the output dataframe row. Initial experiments indicate that the scan operation takes 0.07s per row.

To overcome this problem, a more efficient implementation makes use of hash tables. Specifically, two hash tables with the same structure are generated, one for each input dataframe, having, as key, a hash obtained from each row and, as value, the original index of the row (Fig. 15). To derive the provenance, the output dataframe D is then decomposed into two dataframes obtained by projecting D on the columns of the input ones. Then, the two dataframes so obtained are hashed using the same function above (Fig. 16). This allows us to derive easily the provenance of each row of the join as shown in Fig. 17.

7 Evaluation

All experiments illustrated in this section were performed on a MacBook Pro with 2.6 GHz Intel Core i7 6 core and 32GB RAM 2400MHz. We focus our evaluation on pandas operators for data cleaning and pre-processing, and can theoretically accommodate ML libraries such as scikit-learn as shown in Table 1 although our reference implementation does not explore using their libraries. Given the reference prototype nature of the implementation, the evaluation does not address scalability and performance requirements of a production-grade system.

7.1 Analysis with real world pipelines

Datasets. In Table 2 reported at page 8 we have shown classic provenance queries in terms of data input and output. In order to evaluate if we can answer those queries, we have captured data provenance in three real world pipelines involving different types of preprocessing steps. The datasets are described in Table 3.

Table 3: Datasets used for evaluation.

	German Credit	Compas Score	Census
Records	1000	7214	32561
Features	21	53	15
# Operations	4	7	5
Output Records	1000	6907	32561
Output Features	60	8	104
Provenance Entities	85000	349970	3874264
Provenance Activities	26	7	20
Provenance Relations	255000	451412	9703396

The goal of the *German Credit* pipeline is to predict whether an individual is a good lending candidate. On the other hand, the *Compas Score* pipeline is aimed at predicting the recidivism risk of an individual, whereas the goal of the *Census* pipeline is to predict whether annual income for an individual exceeds \$50K. Table 4 shows the preprocessing steps for each of these machine learning pipelines.

Capturing provenance. Our work focuses on fine-grained provenance and, as such, it turned out that all provenance queries in Table 2 were answerable.

Figure 18 shows the impact of adding provenance capture to a pipeline. The percentage of overhead of capturing provenance is large compared to executing the system without any provenance at all. However, the actual time to capture the provenance itself is rather low: 1.8s for German Credit, 1.4s for COMPAS, and 28s for Census. These results are 10x faster than the times required for the generation of the same provenance in [31]. This improvement is mainly due to the new format used to represent the provenance, and the backend used to store the data, as described in Section 6. As expected, provenance capture adds computational time to any pipeline execution. However, we note that there are certain complex operations that have a larger impact than others. For instance, in the Census pipeline, the generation of the provenance for operation C2 (One Hot encoding of 7 different columns) requires 22ms. However, this operation introduces 90 new features while the number of records remains unchanged (32,561). Therefore, it generates $32,561 \times 90$ new provenance entities. Similarly, operation A3 in the German Credit pipeline is a One-Hot encoding that operates over 11 different columns and creates 38 new features. It follows that this operation creates $1,000 \times 38$ new provenance records. Operation B0 in the Compass Score pipeline, which selects 9 columns of data and removes 44 features, is also costly as it generates $7,214 \times 44$ provenance records. Basically, all the other operations generate a limited amount of data provenance and for this reason, they introduce a limited overhead. The size of provenance generated for the various pipelines is as follows: German Credit 20 MB; Compas Score 71 MB; Census 1.04 GB.

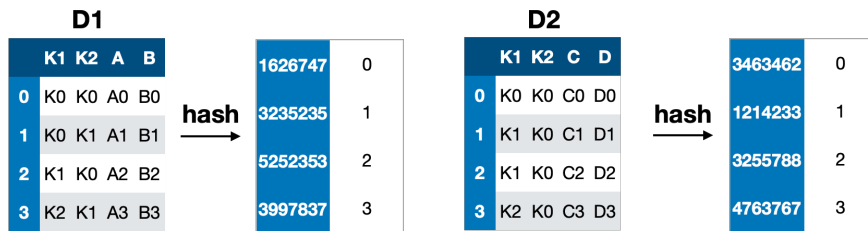


Figure 15: Hashing of the input dataframes

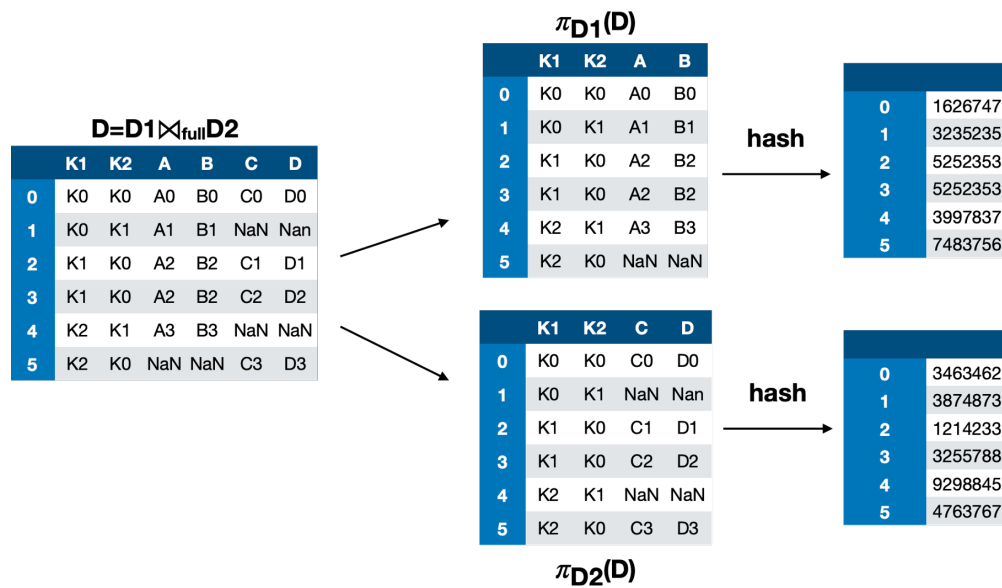


Figure 16: Hashing of output dataframe

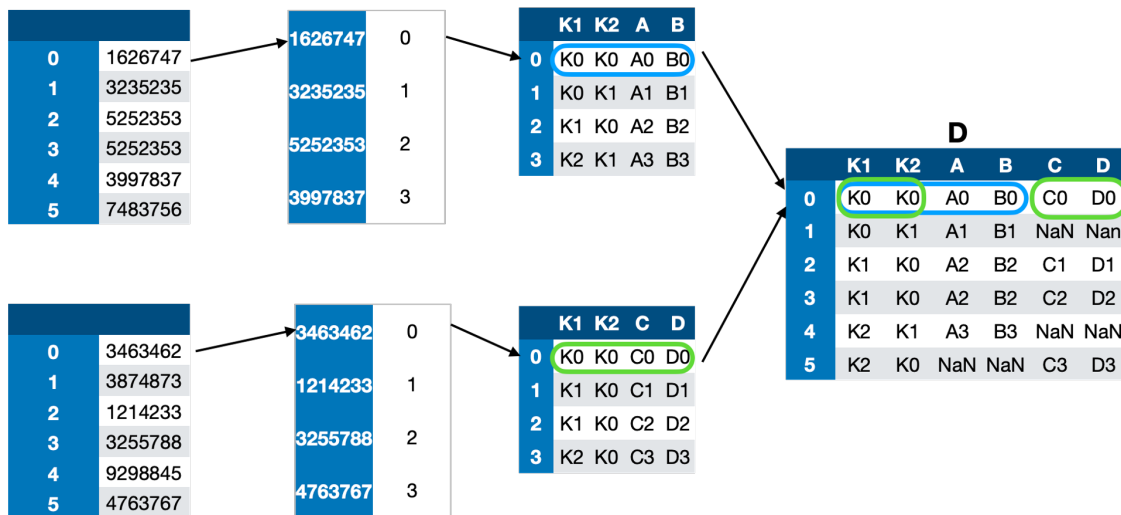


Figure 17: Provenance derivation for the join

Table 4: The preprocessing operations included in the machine learning pipelines used in the evaluation.

German Credit		Compas Score		Census	
Op	Description	Op	Description	Op	Description
A0	Value transformation of 13 distinct columns from codes to interpretable terms.	B0	Selection of 9 relevant columns.	C0	Remove whitespace from 9 columns.
A1	Generation of two new columns from the column <i>personal_status</i> .	B1	Missing values were deleted.	C1	Replace '?' charater for NaN value.
A2	The column <i>personal_status</i> was deleted.	B2	The column <i>race</i> was binarized.	C2	7 categorical columns were OneHot encoded.
A3	11 categorical columns were OneHot encoded.	B3	Value transformation of the <i>label</i> column for consistency.	C3	Two columns were binarized.
		B4	Conversion of <i>c_jail_in</i> and <i>c_jail_out</i> columns to days.	C4	<i>fnlwgt</i> column was deleted.
		B5	Drop <i>jail_in</i> and <i>jail_out</i> dates.		
		B6	Value transformation of column <i>c_charge_degree</i> .		

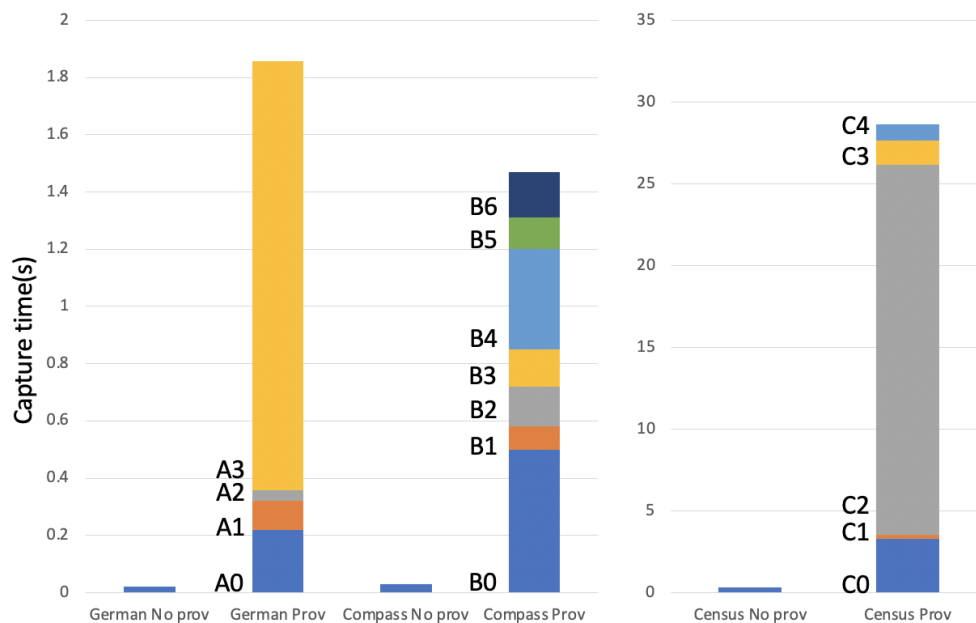


Figure 18: Comparison of cumulative provenance capture times, broken down by individual operator.

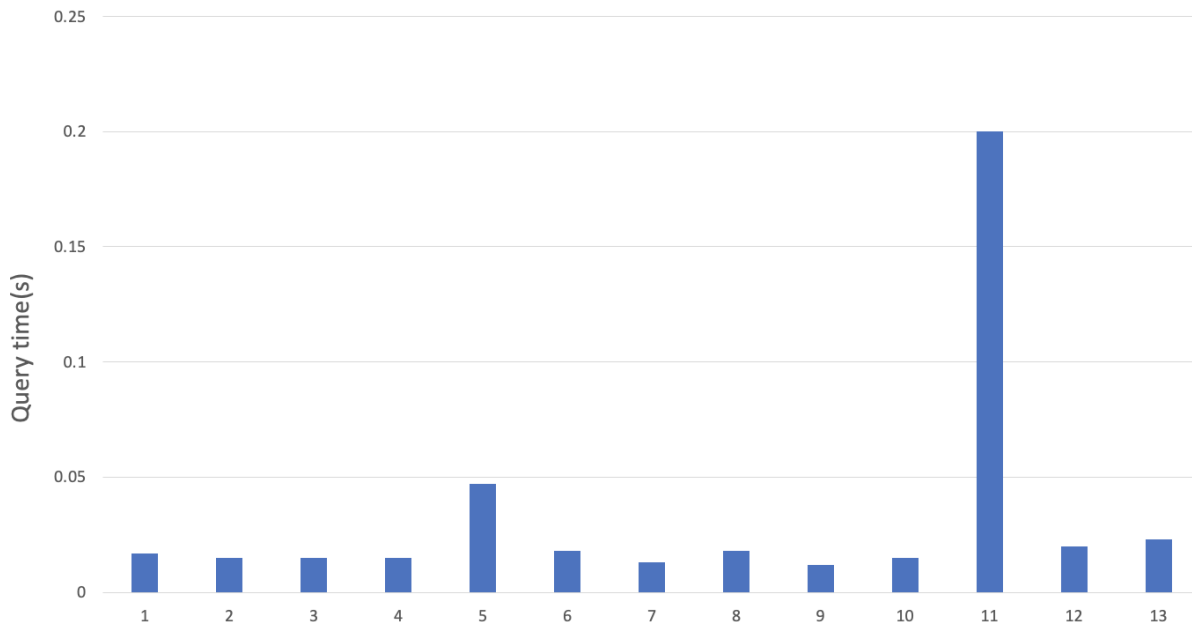


Figure 19: The provenance query times for each type of provenance query are shown in Table 2.

Table 5: Datasets created with the DIGen generator

For Test	Dataset	Scale Factor	Records	Features	Size
Basic	Dataset 1	3	390978	45	5.2 GB
	Dataset 2	5	650412	45	8.6 GB
	Dataset 3	9	1171107	45	16 GB
Join	Dataset 4	3	390978/362342	14/5	349 MB/117 MB
	Dataset 5	5	650412/602956	14/5	582 MB/195 MB
	Dataset 6	9	1171107/1085239	14/5	1,05 GB/342 MB
Append	Dataset 7	3	31581/66689	17/17	38 MB/81 MB
	Dataset 8	5	55650/138889	17/17	68 MB/170 MB
	Dataset 9	9	109034/283298	17/17	134 MB/348 MB

Querying Provenance. Provenance would be useless without the ability to query it efficiently. For this, we run all the types of queries reported in Table 2 over the Census dataset, expressing them in Cypher, the query language of Neo4j. Each query was run three times and the resulting time is the average of the three runs. Queries 2 through 6 operate over a single item, a single record, or a single feature, while the others operate over the entire dataset. For the former type of query, data items, records, and features have been chosen randomly from the output dataset each time the query is run.

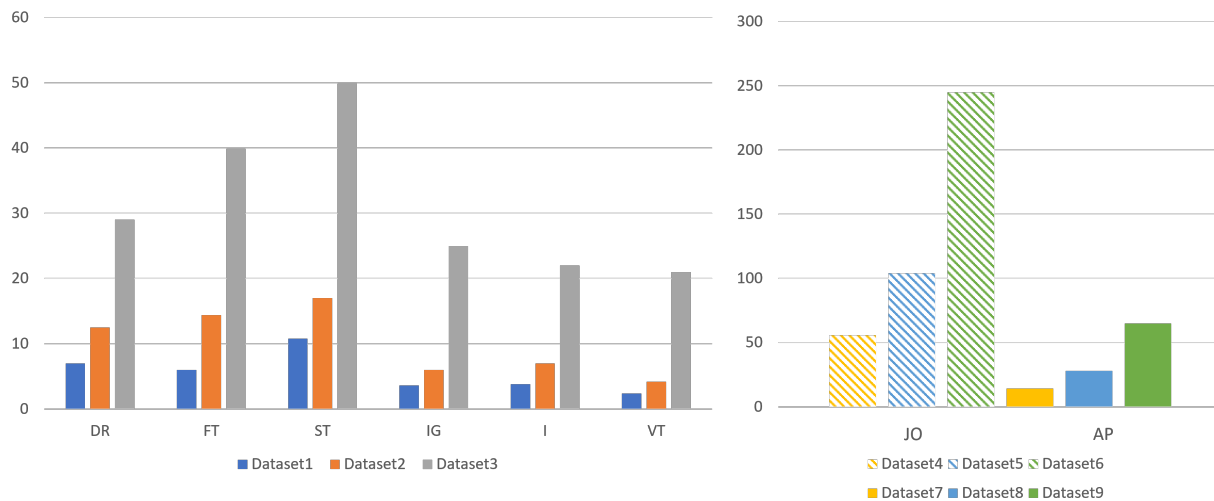
As shown in Figure 19, the basic provenance queries, particularly those that find or trace paths, are fast. The high-cost queries, such as Query 11, require additional processing beyond graph traversal. Recall from Section 2.5 that queries 10 and 11 traverse the provenance graph *and* search for all future and past derivations of an element. Obviously, depending on the complexity of the operations, the operation can require longer time.

7.2 A closer look at operators

We use the TPC-DI benchmark [12] and used DIGen, the data generator provided by TPC, for creating source data and audit information in order to create a known dataset at a larger scale to characterize the behaviour of the reference implementation across a wider range of operators, including Joins and Appends. Specifically, we have created datasets of increasing sizes as described in Table 5: the datasets 1, 2, and 3 involve the trade fact table and the account dimension table, and have been used to measure the effect of unary operators in Table 6 (DR, FT, ST, IG, VT). Datasets 4, 5, and 6 involve the trade.txt and HoldingHistory.txt files and were used to measure the effect of the join operator (JO in Table 6). Datasets 7,8,9 involve the FINWIRE files and were used to measure the effect of the append operator (AP in Table 6).

Table 6: The operations performed on the TCI-DI datasets to test each provenance template.

Op. ID	Operation	Description
DR	Dimensionality Reduction	A column (D_{*j}) is removed from the initial dataset.
FT	Feature Transformation	Transformation on C_GNDR column. Values of gender column are corrected.
I	Imputation	Imputation on T_COMM column. Null values of trade price column are filled with the average value of the column.
ST	Space Transformation	A new column with boolean values is added. 0 if commission value is null, 1 otherwise.
IG	Instance Generation	Generation of one new record.
VT	Value Transformation	Value transformation on C_DOB column. Invalid date of birth are replaced with NaN values.
JO	JOin	Left outer join between Trade table and Holding History on Trade ID
AP	APpend	FINWIRE files from 1967 to 1993 and from 1994 to 2017 were concatenated and then used to perform an append



Operation	Dataset 1	Dataset 2	Dataset 3
Dimensionality Reduction	10 MB	17 MB	31 MB
Feature Transformation	70 MB	120 MB	218 MB
Imputation	15 MB	25 MB	46 MB
Space Transformation	60 MB	101 MB	183 MB
Instance Generation	10 MB	17 MB	31 MB
Value Transformation	110 KB	400 KB	550 KB
Join	1.02 GB	1,78 GB	3.04 GB
Append	390 MB	779 MB	1,6 GB

Figure 20: Capture time (in seconds) and storage space for each operation

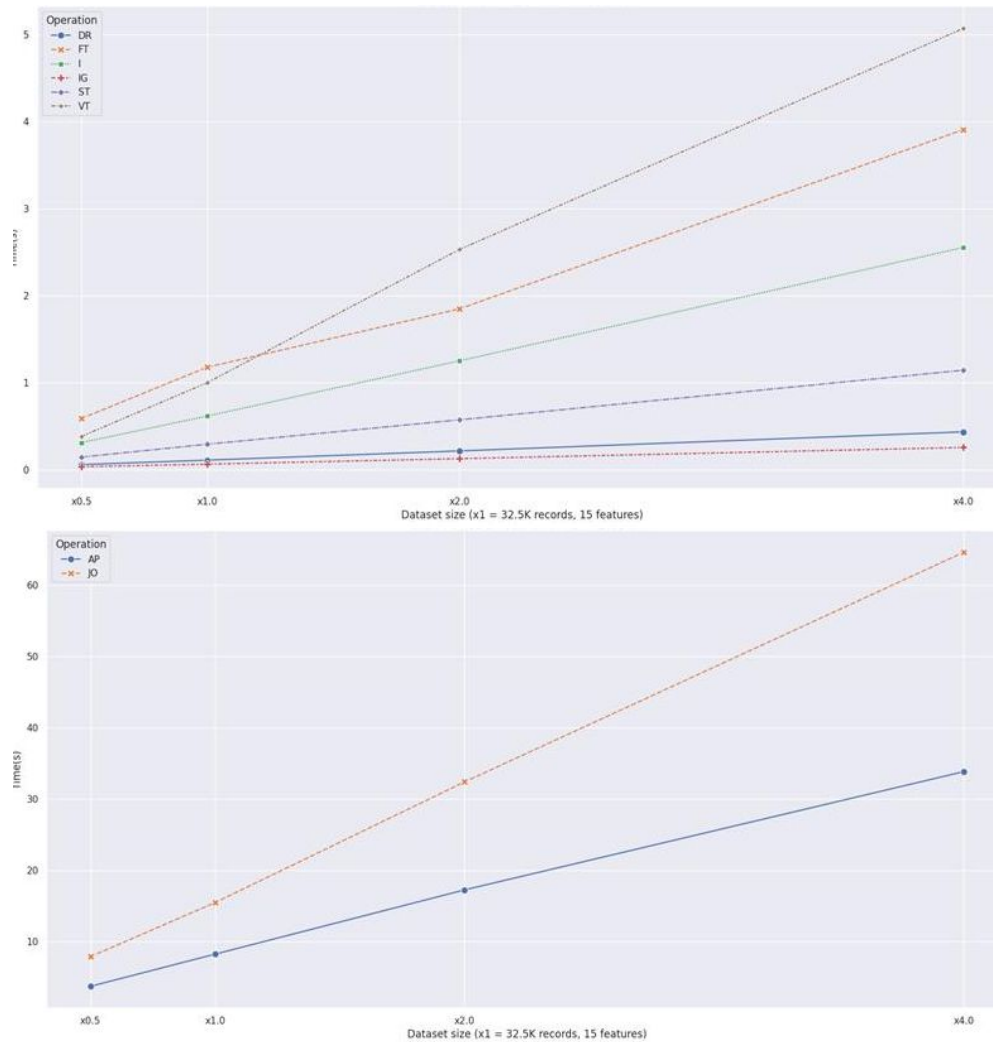


Figure 21: Capture time (in seconds) by varying the size of a fixed dataset

Figure 20 shows how long and how much space it takes to capture and record provenance for each operation. The capture mechanism scales rather well with the size of the dataset and it turns out that pre-processing operations that only affect a small number of data values, such as Instance Generation (IG), are fast. Value Transform (VT) and Imputation (I), in this particular evaluation setup, are also fast as they only operate over a small number of items. On the other hand, the operations that generate more provenance, such as Feature Transformation (FT), Space Transformation (ST), and Dimensionality Reduction (DR), take more time. In particular, ST needs to create provenance data for every new value in the new column. Join (JO) and Append (AP) operations require more time as they need to generate a quite large quantity of provenance. In addition, JO is more costly as it operates over two input tables.

The fact that provenance capture scales gracefully with the dataset size is confirmed by another experiment, whose results are reported in Figure 21, in which we have executed the same types of operations in Table 6 by just varying the number of records of a fixed input (the Census dataset). Since the evaluation setup here is different, the various operators exhibit a different behavior in terms of relative performances, but their computational time remains quite low and grows linearly with the dataset size.

Table 7: Issues identified in real Machine Learning pipelines and provenance queries that can provide support to them.

Id	Data Science Stack Exchange Use Cases	Prov. query id
UC1	When applying the Predictions widget on the same training dataset, the results (i.e. probability scores) are different: https://datascience.stackexchange.com/questions/32382/orange-predictions-widget-on-same-data-gives-different-results	PQ1
UC2	Differences in the predictions and goodness-of-fit of R2 metric for the linear regression model on Orange and Scikit-learn: https://datascience.stackexchange.com/questions/32678/orange-linear-regression-and-scikit-learn-linear-regression-gives-different-resu	PQ2
UC3	After performing image classification using an ML model, prediction probabilities are constant on test images https://datascience.stackexchange.com/questions/38320/orange3-image-classification	PQ3
UC4	From a constructed workflow using image classification (add-on widgets) ascertain whether the workflow performs transfer learning: https://datascience.stackexchange.com/questions/19240/using-orange3-to-predict-image-class	PQ3
UC5	Application of the Test and Score and Predictions widget on the same data utilising the same ML model; produces differing results: https://datascience.stackexchange.com/questions/20572/why-orange-predictions-and-test-score-produce-different-results-on-the-sam	PQ3
UC6	When applying the Impute widget during preprocessing on the train/test dataset, the same values are predicted for all rows: https://datascience.stackexchange.com/questions/15264/orange-3-same-prediction-for-all-of-my-data-when-using-impute-widget	PQ4, PQ5, PQ6, PQ11, PQ12
UC7	Inaccuracy in the prediction of target variable using k-NN and linear regression ML models in an Orange workflow: https://datascience.stackexchange.com/questions/36537/how-to-properly-predict-date-using-orange-3	PQ7, PQ8, PQ9, PQ10
UC8	Disproportionate allocation of labels after performing data analysis and modeling (inaccurate classification accuracy): https://datascience.stackexchange.com/questions/37471/dataset-with-disproportionately-more-of-a-single-label-than-any-other	PQ11, PQ12

7.3 Use Case Analysis

Table 7 contains a collection of real-world scenarios in which data scientists try to understand what is happening within a machine-learning pipeline. These use cases have been gathered from the Data Science Stack Exchange⁴ (DSSE) by selecting questions about the construction of a data preparation pipeline using the Orange framework. The provenance queries that can provide support to these issues refer to those in Table 2 (Page 8), in which, for each query, it is reported the input data and the expected output that can help the developer to debug the pipeline.

To highlight how the fine-grained provenance captured with our approach can be used to answer one of these questions consider, for instance, the UC8 use case. In this scenario, the user is struggling with an incorrect high accuracy of a model. Ultimately, this is because of an imbalanced input dataset. Using the Provenance Query *Impact on Feature Spread* from Table 2 on the input dataset, it is possible to identify the change of feature spread after a pre-processing operator that rebalances the dataset.

7.4 Provenance Exploration

Unlike many provenance systems, which focus on the presentation and navigation of the provenance graph, we have developed a tool⁵ in which the provenance graph is mainly used as a backbone to explore and identify problems within the pipeline through a user-friendly interface, which does not require the specification of complex queries over the data provenance. This is done by automatically extracting, from the provenance and other metadata, useful information on the changes operated by the individual operations on the input dataset(s).

An example of this kind of interaction is shown in the GUIs of our tool reported in Figures 22 and 23 respectively: basically, depending on the type of operator that was applied, the data scientist can “zoom in” to a transformation of

⁴<https://datascience.stackexchange.com/>

⁵The code of this tool is publicly available on GitHub.

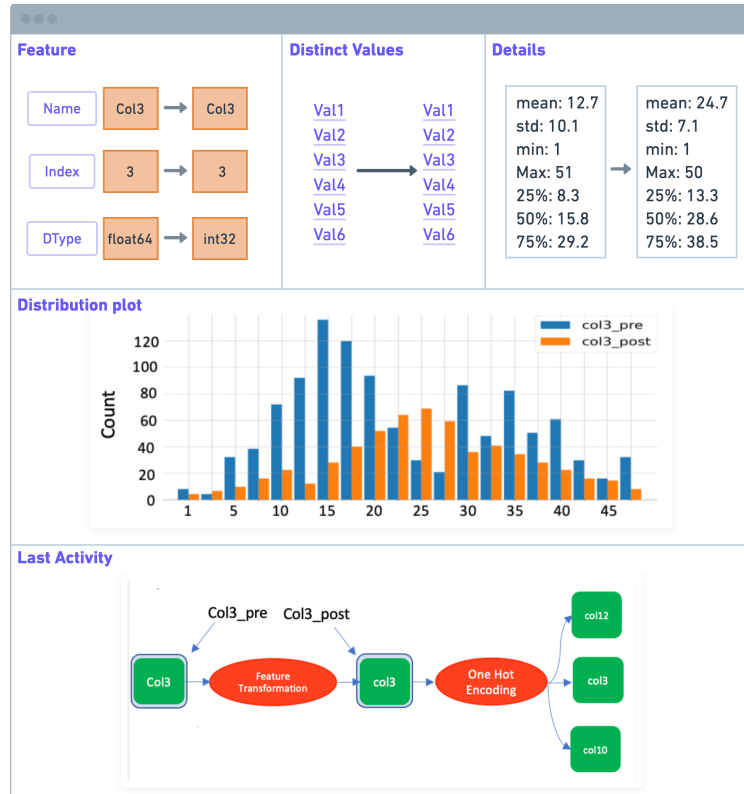


Figure 22: Example of a GUI showing how data changes after a transformation process that operates locally, at the feature level.

interest (bottom of the figures) and inspect the “before/after” effect of its execution either at the level of values within a column, in the case of a *local* transformation, or at the level of the entire dataset, in the case of a *global* transformation.

A local type of transformation is illustrated in Figure 22, where a data transformation operation, which modifies values in COL3, is represented in the provenance fragment at the bottom. The user is then able to navigate through the retrieved provenance, identify the pre- and post- states for COL3, and visualise their differences in terms of summary statistics (top right), values distribution (center), and optionally each value can be inspected (top middle).

In contrast, Figure 23 shows the effect of an imputation step that operates globally. As this may change more than one column at a time (for instance, using Multiple Inference), here the GUI displays salient differences at the dataset level. We can see for instance that the operation has not changed the number of rows and columns of the dataset (top left), but the imputation has updated the content of several columns (col2, col4, col5, col6, see top middle), and has altered the percentage of null values (bar chart). We can also see the changes in the correlation between each pair of columns in the dataset before and after the operation is performed.

7.5 Comparison to other provenance collection systems

There are many provenance systems that can be deployed to capture provenance of workflow-like executions. In this section, we look at some of the main players and compare them to the provenance in this work. Because the implementation in this work was a reference implementation for exploration, not production deployment, we feel that an execution benchmark between the systems is uninteresting.

Perm [43, 41]. Perm uses query rewrites to add and propagate provenance attributes to the output of the original query. It can capture and propagate provenance for ASPJ queries and set operations. It is implemented on PostgreSQL and tested against TPC-H with an overhead on TPC-H queries of 3-4x. The queries outside of this norm include very complex queries with aggregation, such as an aggregation over a join on 8 tables with a grouping on a functional expression. Perm allows lazy and eager computation of provenance, SQL query facilities, and support for external provenance. Perm outperformed previous approaches by a factor of 30. While the execution of Perm is impressive, it fundamentally relies on a technology that is not appropriate for the problem within notebooks focused on in this work.

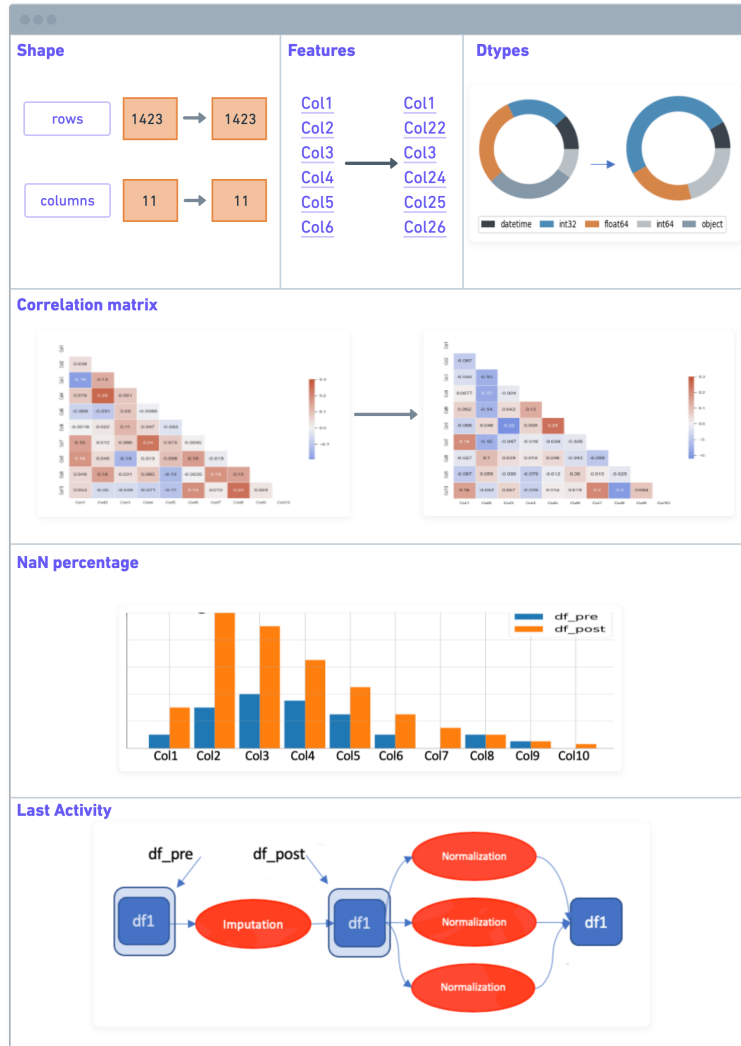


Figure 23: Example of a GUI showing how data changes at the dataset level, after an imputation operation over several columns.

MLInspect [36, 38]. The *MLInspect* system uses Python’s inspect module or Monkey patch to identify function calls within python scripts and build a DAG of relationships and interactions in the pre-processing pipeline. This run-time representation is updated as the developer changes the scripts based on the standard dataframe operators. A user can annotate tuples, and specify the inspections that need to occur (e.g. inspect for statistical parity of protected group). As the script is executed, this DAG is stepped through, and the operator is passed for inspection. The focus of the *MLInspect* is to analyze data distribution after operators based on the pre-specified inspections. Provenance at the tuple level is supported through the adaptation of user annotations by recording the pre-assigned tuple-id and the operator applied. Our work has a very different focus, and as such the provenance requirements are different. In *MLInspect*, provenance can be added at the tuple level and used to support data distribution change analysis; our work provides a much finer-grained provenance at the attribute level allowing for debugging specific value changes.

Vamsa [42]. The *Vamsa* system uses static analysis to build a syntax tree to identify inputs, parameters and libraries. It then uses a knowledge base to provide semantic meaning to these items and log it in the provenance. However, *Vamsa* relies upon a pre-populated knowledge base which maps the set of functions identified in the code to semantic "operators". This approach restricts provenance generation to known operators. The *Vamsa* experimentation shows that their coverage ranges between 74.48% - 97.08%. Our approach relies on inspection of the dataframe before and after to identify the type of transformation that occurred (e.g. horizontal reduction) instead of relying upon a pre-created knowledge base. In addition to this collection difference, *Vamsa* identifies and store provenance information at the dataset level. For instance, it will identify entire rows retained or dropped; it can identify whether a column

Table 8: Comparison of this work with other provenance capture systems

Provenance System	Deployed on	Capture Method	Granularity
This work	Jupyter Notebooks with dataframes	Operator derived from comparison of before-after dataframes; provenance templates associated with operator.	attribute
MLInspect [36][38][39]	dataframes	Python’s inspect module and Monkey patch to identify operators at execution to build a DAG of operators	tuple
Perm [40, 41]	Relational databases	Query rewrite to add and propagate provenance attributes to the output of the original query.	tuple
Vamsa [42]	python scrips with dataframes	Static analysis of script to identify inputs, parameters and libraries; knowledge base to provide semantic meaning.	dataset

within a dataframe is used. However, it does not track individual changes to attributes. While these can be later derived by understanding what operators were applied to which rows or columns, this information is not innately stored. A combination of Vamsa and this work would be interesting future work, in which Vamsa is used to identify the pre-stocked operators, and for the remainder, DPDS identifies what is happening in the via dataframe changes and templates of provenance.

8 Related Work

This paper substantially advances previous work [31] by: (i) extending the set of core operations with methods for combining different datasets to any operator that modifies a dataframe, (ii) replacing the manual instrumentation at the script level required by the analysts with a method for the identification of provenance for most of the operators through dataset change (iii) adopting a graph-based data management system for storing and querying in an effective and efficient way the collected provenance, (iv) performing experiments for empirical validation and qualitative comparison to previous work.

Established techniques and tools are available to generate provenance, and *provenance polynomials* through query instrumentation. However, these operate in a relational database setting and assume that queries use relational operators [44, 45, 41]. While we show how some of the pipeline operators considered in this work map to relational algebra, this is not true for all of them, so we prefer to avoid techniques that are tightly linked to SQL or to first-order queries [46] as these would preclude other types of operators from being included in the future. We, therefore, consider this an unwise strategy in an “open world” of data pre-processing operators, consider e.g. *one-hot* and other kinds of categorical data encodings. We also note that tools that operate on a database back-end, like *GProm* [44], *Smoke* [15] and older ones like *Post-it* [47] for provenance capture cannot be used in our setting. Interestingly, extensions to the polynomials approach have been proposed to describe the provenance of certain linear algebra operations, such as matrix decomposition and tensor-product construction [48]. While these can potentially be useful, it is a partially developed theory with limited and specialised applicability.

Moving beyond relational data provenance, capturing provenance within scripts is also not new, but efforts have mostly focused on the provenance of script definition, deployment, and execution [49]. Specifically, a number of tools are available to help developers build machine learning pipelines [50, 18, 51] or debug them [52], but these lack the ability to explain the provenance of a certain data item in the processed dataset. Others link provenance to explainability in a distributed machine learning setting [53] but without offering specific tools. Amazon identifies that there are common and reusable components to a machine learning pipeline, but that there is no way to track the exploration of pipeline construction effectively, and calls for metadata capture to support reasoning over pipeline design [54]. Vamsa [42] attempts to tackle some of these problems by gathering the provenance of pipeline design. However, the resulting provenance documents contain information such as the invocation of specific ML libraries, by way of automated script analysis, rather than data derivations. Some systems are designed to help debug ML pipelines. BugDoc [55] looks at changes in a pre-processing pipeline that cause the models to fail, where high-level script and orders are used to identify bad configurations. Others provide quality assurance frameworks [56] or embedded simulators to estimate the fairness impacts of a particular pipeline [57]. Again, however, these are not geared for deep data introspection. Priu [58], helps users understand data changes, particularly deletions, that are used in regression models. Unfortunately, this work only tracks deletions and not additions or updates to data.

Recently, [8] have utilized provenance to understand the changes in data distribution in the ML pipeline using predefined “inspections” that look at the data at specific operators within the pipeline, which supports the reason for undertaking this work and which we expand by unobtrusively capturing provenance from any operator. Meanwhile, [30] combines system level provenance information with application-level log files to recreate the provenance of data science pipelines without impacting the pipeline developer. Other tools record the execution of generic (python) scripts, but fail to capture detailed data provenance, like NoWorkflow [59, 33]. This has been combined with YesWorkflow [60, 35] which provides a workflow-like description of scripts, but again without a focus on data derivations.

A further class of tools instrument scripts that are specifically designed for Big Data processing frameworks: [61] (Hadoop), [62, 63, 64, 15] (Spark). They provide detailed information mostly for debugging purposes but are restricted in their scope of applicability.

Recently, a method for fine-grained provenance capture that is application-agnostic has been proposed [30]. Here, provenance from the low-level OS through to high-level application-specific logs is merged to create a provenance record that contains the maximum information available for the minimum impact on developers. However, it is not obvious what fine-grain provenance can be extracted from such an approach, while our work provides a firm basis for the provenance information that should be captured. Interesting future work includes determining how much of the provenance we specify can be collected by [30].

Finally, the method proposed within Section 5 in which the change of the data is observed instead of the operator is similar to techniques discussed in [26]. While Blount describes the general setup of inferring the provenance record based on identified changes in the data, our work provides a functioning implementation for a large class of operators.

9 Conclusions and Future Work

In this work, we focus on fine-grained data provenance for machine learning pipelines irrespective of the pipeline tool used. Because a substantial effort goes into selecting and preparing data for use in modelling, and because changes made during preparation can affect the ultimate model, it is important to be able to trace what is happening to the data at a fine-grain level.

We highlight several real use cases to motivate the need for fine-grained provenance from the Data Science Stack Exchange (DSSE)¹. We identify the classic provenance queries that are needed to provide information to answer these use cases. We then identify a set of provenance templates that can be deployed across a set of machine learning pipeline operators and implement them.

We depart significantly in this work from previous implementations within python and ML environments, by using *observed* changes in the data to determine the provenance. Based on observations of the changes between dataframes, we choose the appropriate template for provenance generation. We have tested our implementation over real-world ML benchmark pipelines for utility and basic performance with both classic ML pipelines and TCP-DI. Our results indicate that we can collect fine-grained provenance that is both useful and performant.

Future investigation into optimization techniques that aim at reducing the provenance data, using composite generation, to the minimum that is needed to support given provenance queries, as well as methods for taking advantage of collected provenance data to support the design of new pipelines is required to continue making provenance more efficient and useful. This work looks expressly at the pre-processing tools leading up to the machine learning black box, thus it does not track provenance models for the trained data, e.g. between predictions and training data. However, this work has been used by [65] to create an entire tracking of data from pre-processing through deep learning. Future work in this area includes understanding the granularity of provenance required for users of deep learning systems.

References

- [1] Nikolaos Konstantinou, Martin Koehler, Edward Abel, Cristina Civili, Bernd Neumayr, Emanuel Sallinger, Alvaro A.A. Fernandes, Georg Gottlob, John A. Keane, Leonid Libkin, and et al. The vada architecture for cost-effective data wrangling. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD '17*, page 1599–1602, New York, NY, USA, 2017. Association for Computing Machinery.
- [2] Amirata Ghorbani and James Y. Zou. Data shapley: Equitable valuation of data for machine learning. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 2242–2251. PMLR, 2019.

- [3] Arvind Narayanan. Translation tutorial: 21 fairness definitions and their politics. In *Proc. Conf. Fairness Accountability Transp., New York, USA*, 2018.
- [4] Ramaravind Kommiya Mothilal, Amit Sharma, and Chenhao Tan. Explaining machine learning classifiers through diverse counterfactual explanations. *arXiv preprint arXiv:1905.07697*, 2019.
- [5] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. “why should I trust you”: Explaining the predictions of any classifier. In Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi, editors, *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 1135–1144. ACM, 2016.
- [6] Himabindu Lakkaraju, Ece Kamar, Rich Caruana, and Jure Leskovec. Interpretable & explorable approximations of black box models. *CoRR*, abs/1707.01154, 2017.
- [7] Ahmed M Alaa and Mihaela van der Schaar. Demystifying black-box models with symbolic metamodels. In *Advances in Neural Information Processing Systems*, pages 11301–11311. Curran Associates, Inc., 2019.
- [8] Stefan Grafberger, Paul Groth, Julia Stoyanovich, and Sebastian Schelter. Data distribution debugging in machine learning pipelines. *The VLDB Journal*, pages 1–24, 2022.
- [9] Luc Moreau, Paolo Missier, Khalid Belhajjame, Reza B’Far, James Cheney, Sam Coppens, Stephen Cresswell, Yolanda Gil, Paul Groth, Graham Klyne, Timothy Lebo, Jim McCusker, Simon Miles, James Myers, Satya Sahoo, and Curt Tilmes. PROV-DM: The PROV Data Model. Technical report, World Wide Web Consortium, 2012.
- [10] Salvador García, Sergio Ramírez-Gallego, Julián Luengo, José Manuel Benítez, and Francisco Herrera. Big data preprocessing: methods and prospects. *Big Data Analytics*, 1(1):9, dec 2016.
- [11] Bilal Mirza, Wei Wang, Jie Wang, Howard Choi, Neo Christopher Chung, and Peipei Ping. Machine Learning and Integrative Analysis of Biomedical Big Data. *Genes*, 10(2):87, jan 2019.
- [12] Meikel Poess, Tilmann Rabl, Hans-Arno Jacobsen, and Brian Caulfield. Tpc-di: The first industry benchmark for data integration. *Proc. VLDB Endow.*, 7(13):1367–1378, August 2014.
- [13] Alvin Cheung. *Rethinking the Application-Database Interface*. PhD thesis, Massachusetts Institute of Technology, 2015.
- [14] Devin Petersohn, William W. Ma, Doris Jung Lin Lee, Stephen Macke, Doris Xin, Xiangxi Mo, Joseph Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya G. Parameswaran. Towards scalable dataframe systems. *Proc. VLDB Endow.*, 13(11):2033–2046, 2020.
- [15] Fotis Psallidas and Eugene Wu. Smoke: Fine-grained lineage at interactive speed. *Proceedings of the VLDB Endowment*, 11(6):719–732, 2018.
- [16] Janez Demšar, Tomaž Curk, Aleš Erjavec, Črt Gorup, Tomaž Hočevar, Mitar Milutinovič, Martin Možina, Matija Polajnar, Marko Toplak, Anže Starič, Miha Štajdohar, Lan Umek, Lan Žagar, Jure Žbontar, Marinka Žitnik, and Blaž Zupan. Orange: Data mining toolbox in python. *Journal of Machine Learning Research*, 14:2349–2353, 2013.
- [17] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [18] Janez Demšar, Tomaž Curk, Aleš Erjavec, Črt Gorup, Tomaž Hočevar, Mitar Milutinovič, Martin Možina, Matija Polajnar, Marko Toplak, Anže Starič, et al. Orange: data mining toolbox in python. *The Journal of Machine Learning Research*, 14(1):2349–2353, 2013.
- [19] Nan Zheng, Abdussalam Alawini, and Zachary Ives. Fine-grained provenance for matching & etl. *Proceedings. International Conference on Data Engineering*, 2019:184–195, 04 2019.
- [20] Luc Moreau, James Cheney, and Paolo Missier. Constraints of the prov data model, 2013.
- [21] Luca Cabibbo and Riccardo Torlone. A logical approach to multidimensional databases. In Hans-Jörg Schek, Fèlix Saltor, Isidro Ramos, and Gustavo Alonso, editors, *Advances in Database Technology - EDBT’98, 6th International Conference on Extending Database Technology, Valencia, Spain, March 23-27, 1998, Proceedings*, volume 1377 of *Lecture Notes in Computer Science*, pages 183–197. Springer, 1998.
- [22] P Buneman, S Khanna, and WC Tan. Why and where: A characterization of data provenance. In VanDenBussche, J and Vianu, V, editor, *DATABASE THEORY - ICDT 2001, PROCEEDINGS*, volume 1973 of *Lecture Notes in Computer Science*, pages 316–330. SPRINGER-VERLAG BERLIN, 2001.
- [23] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. Provenance in databases: Why, how, and where. *Found. Trends Databases*, 1(4):379–474, apr 2009.

- [24] Adriane Chapman and H. V. Jagadish. Why not? In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, page 523–534, New York, NY, USA, 2009. Association for Computing Machinery.
- [25] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '07, page 31–40, New York, NY, USA, 2007. Association for Computing Machinery.
- [26] Tom Blount, Adriane Chapman, Michael Johnson, and Bertram Ludascher. Observed vs. possible provenance. In *13th International Workshop on Theory and Practice of Provenance (TaPP 2021)*, 2021.
- [27] Trivellore E Raghunathan, James M Lepkowski, John Van Hoewyk, Peter Solenberger, et al. A multivariate technique for multiply imputing missing values using a sequence of regression models. *Survey methodology*, 27(1):85–96, 2001.
- [28] Yael Amsterdamer, Susan B. Davidson, Daniel Deutch, Tova Milo, Julia Stoyanovich, and Val Tannen. Putting lipstick on pig: Enabling database-style workflow provenance. *CoRR*, abs/1201.0231, 2012.
- [29] Sarah Cohen-Boulakia, Olivier Biton, Shirley Cohen, and Susan Davidson. Addressing the provenance challenge using zoom. *Concurr. Comput.*, 20(5):497–506, apr 2008.
- [30] Lukas Rupprecht, James C. Davis, Constantine Arnold, Yaniv Gur, and Deepavali Bhagwat. Improving reproducibility of data science pipelines through transparent provenance capture. *Proc. VLDB Endow.*, 13(12):3354–3368, August 2020.
- [31] Anonymous authors. Capturing and querying fine-grained provenance of preprocessing pipelines in data science. *Proceedings of the VLDB Endowment*, 2020.
- [32] M David Allen, Adriane Chapman, Barbara Blaustein, and Len Seligman. Capturing provenance in the wild. In *Provenance and Annotation of Data and Processes: Third International Provenance and Annotation Workshop, IPAW 2010, Troy, NY, USA, June 15-16, 2010. Revised Selected Papers 3*, pages 98–101. Springer, 2010.
- [33] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. noworkflow: a tool for collecting, analyzing, and managing provenance from python scripts. *Proc. VLDB Endow.*, 10(12):1841–1844, 2017.
- [34] Barbara Lerner, Emery R Boose, Orenna Brand, Aaron M Ellison, Elizabeth Fong, Matthew K Lau, Khanh Ngo, Thomas Pasquier, Luis Perez, Margo I Seltzer, et al. Making provenance work for you. *R J.*, 14(4):141–159, 2023.
- [35] Qian Zhang, Paul J Morris, Timothy McPhillips, James Hanken, David Lowery, Bertram Ludäscher, James Macklin, Robert Morris, and John Wieczorek. Using yesworkflow hybrid queries to reveal data lineage from data curation activities. *Biodiversity Information Science and Standards*, 1:e20380, 2017.
- [36] Stefan Grafberger, Shubha Guha, Julia Stoyanovich, and Sebastian Schelter. Mlinsect: A data distribution debugger for machine learning pipelines. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2736–2739, New York, NY, USA, 2021. Association for Computing Machinery.
- [37] Trung Dong Huynh. *Prov python*, 2018.
- [38] Stefan Grafberger, Paul Groth, Julia Stoyanovich, and Sebastian Schelter. Data distribution debugging in machine learning pipelines. *The VLDB Journal*, 31:1103—1126, 2022.
- [39] Stefan Grafberger, Paul Groth, and Sebastian Schelter. Provenance tracking for end-to-end machine learning pipelines. In *Companion Proceedings of the ACM Web Conference 2023*, pages 1512–1512, 2023.
- [40] Boris Glavic and Gustavo Alonso. Perm: Processing provenance and data on the same data model through query rewriting. In *2009 IEEE 25th International Conference on Data Engineering*, pages 174–185. IEEE, 2009.
- [41] Boris Glavic and Gustavo Alonso. Perm: Processing provenance and data on the same data model through query rewriting. In Yannis E. Ioannidis, Dik Lun Lee, and Raymond T. Ng, editors, *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pages 174–185. IEEE Computer Society, 2009.
- [42] Mohammad Hossein Namaki, Avrielia Floratou, Fotis Psallidas, Subru Krishnan, Ashvin Agrawal, Yinghui Wu, Yiwen Zhu, and Markus Weimer. Vamsa: Automated provenance tracking in data science scripts. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, page 1542–1551, New York, NY, USA, 2020. Association for Computing Machinery.
- [43] Boris Glavic and Gustavo Alonso. The perm provenance management system in action. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 1055–1058, 2009.
- [44] Xing Niu, Raghav Kapoor, Boris Glavic, Dieter Gawlick, Zhen Hua Liu, and Venkatesh Radhakrishnan. Provenance-aware query optimization. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 473–484. IEEE Computer Society, 2017.

- [45] Bahareh Sadat Arab, Su Feng, Boris Glavic, Seokki Lee, Xing Niu, and Qitian Zeng. Gprom - A swiss army knife for your provenance needs. *IEEE Data Eng. Bull.*, 41(1):51–62, 2018.
- [46] Seokki Lee, Sven Köhler, Bertram Ludäscher, and Boris Glavic. A sql-middleware unifying why and why-not provenance for first-order queries. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 485–496. IEEE Computer Society, 2017.
- [47] Laura Chiticariu, Wang Chiew Tan, and Gaurav Vijayvargiya. Dbnotes: a post-it system for relational databases based on provenance. In Fatma Özcan, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 942–944. ACM, 2005.
- [48] Zhepeng Yan, Val Tannen, and Zachary G. Ives. Fine-grained provenance for linear algebra operators. In Sarah Cohen Boulakia, editor, *8th USENIX Workshop on the Theory and Practice of Provenance, TaPP 2016, Washington, D.C., USA, June 8-9, 2016*. USENIX Association, 2016.
- [49] João Felipe Pimentel, Juliana Freire, Leonardo Murta, and Vanessa Braganholo. A survey on collecting, managing, and analyzing provenance from scripts. *ACM Computing Surveys (CSUR)*, 52(3):1–38, 2019.
- [50] Pulkit Agrawal, Rajat Arya, Aanchal Bindal, Sandeep Bhatia, Anupriya Gagneja, Joseph Godlewski, Yucheng Low, Timothy Muss, Mudit Manu Paliwal, Sethu Raman, and et al. Data platform for machine learning. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD '19*, page 1803–1816, New York, NY, USA, 2019. Association for Computing Machinery.
- [51] Zeyuan Shang, Emanuel Zraggen, Benedetto Buratti, Ferdinand Kossmann, Philipp Eichmann, Yeounoh Chung, Carsten Binnig, Eli Upfal, and Tim Kraska. Democratizing data science through interactive curation of ml pipelines. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD '19*, page 1171–1188, New York, NY, USA, 2019. Association for Computing Machinery.
- [52] Manasi Vartak, Joana M. F. da Trindade, Samuel Madden, and Matei Zaharia. MISTIQUE: A system to store and query model intermediates for model diagnosis. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1285–1300. ACM, 2018.
- [53] Stefanie Scherzinger, Christin Seifert, and Lena Wiese. The best of both worlds: Challenges in linking provenance and explainability in distributed machine learning. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1620–1629. IEEE, 2019.
- [54] Sebastian Schelter, Joos-Hendrik Böse, Johannes Kirschnick, Thoralf Klein, Stephan Seufert, and Amazon. Declarative metadata management: A missing piece in end-to-end machine learning. In *SysML Conference*, 2018.
- [55] Raoni Lourenço, Juliana Freire, and Dennis Shasha. Bugdoc: Algorithms to debug computational processes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD '20*, page 463–478, New York, NY, USA, 2020. Association for Computing Machinery.
- [56] Stefan Studer, Thanh Binh Bui, Christian Drescher, Alexander Hanuschkin, Ludwig Winkler, Steven Peters, and Klaus-Robert Mueller. Towards crisp-ml (q): A machine learning process model with quality assurance methodology. *arXiv preprint arXiv:2003.05155*, 2020.
- [57] Alexander D’Amour, Hansa Srinivasan, James Atwood, Pallavi Baljekar, D. Sculley, and Yoni Halpern. Fairness is not static: Deeper understanding of long term fairness via simulation studies. In *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency, FAT* '20*, page 525–534, New York, NY, USA, 2020. Association for Computing Machinery.
- [58] Yinjun Wu, Val Tannen, and Susan B. Davidson. Priu: A provenance-based approach for incrementally updating regression models. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 447–462. ACM, 2020.
- [59] João Felipe Pimentel, Juliana Freire, Leonardo Murta, and Vanessa Braganholo. Fine-grained provenance collection over scripts through program slicing. In *International Provenance and Annotation Workshop*, pages 199–203. Springer, 2016.
- [60] Timothy McPhillips, Tianhong Song, Tyler Kolisnik, Steve Aulenbach, Khalid Belhajjame, Kyle Bocinsky, Yang Cao, Fernando Chirigati, Saumen Dey, Juliana Freire, et al. Yesworkflow: a user-oriented, language-independent tool for recovering workflow information from scripts. *arXiv preprint arXiv:1502.02403*, 2015.
- [61] Robert Ikeda, Junsang Cho, Charlie Fang, Semih Salihoglu, Satoshi Torikai, and Jennifer Widom. Provenance-based debugging and drill-down in data-oriented workflows. In Anastasios Kementsietsidis and Marcos Antonio Vaz Salles, editors, *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 1249–1252. IEEE Computer Society, 2012.

- [62] T. Guedes, V. Silva, M. Mattoso, M. V. N. Bedo, and D. de Oliveira. A practical roadmap for provenance capture and data analysis in spark-based scientific workflows. In *Workflows in Support of Large-Scale Science (WORKS)*, pages 31–41. IEEE/ACM, Nov 2018.
- [63] MingJie Tang, Saisai Shao, Weiqing Yang, Yanbo Liang, Yongyang Yu, Bikas Saha, and Dongjoon Hyun. SAC: A system for big data lineage tracking. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 1964–1967. IEEE, 2019.
- [64] Matteo Interlandi, Kshitij Shah, Sai Tetali, Muhammad Gulzar, Seunghyun Yoo, Miryung Kim, Todd Millstein, and Tyson Condie. Titian: Data provenance support in spark. *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, 9:216–227, 01 2016.
- [65] Débora Pina, Adriane Chapman, and and Marta Mattoso Daniel De Oliveira. Deep learning provenance data integration: a practical approach. In *Companion Proceedings of the ACM Web Conference 2023*, 2023.