Logical Provenance in Data-Oriented Workflows*

Robert Ikeda [1], Akash Das Sarma [2], Jennifer Widom [3]

Stanford University, USA

[1]rmikeda@cs.stanford.edu

[2]akashds.iitk@gmail.com [3]widom@cs.stanford.edu

***Abstract*—We consider the problem of defining, generating, and tracing provenance in data-oriented workflows, in which input data sets are processed by a graph of transformations to produce output results. We first give a new general definition of provenance for general transformations, introducing the notions of correctness, precision, and minimality. We then determine when properties such as correctness and minimality carry over from the individual transformations' provenance to the workflow provenance. We describe a simple logical-provenance specification language consisting of attribute mappings and filters. We provide an algorithm for provenance tracing in workflows where logical provenance for each transformation is specified using our language. We consider logical provenance in the relational setting, observing that for a class of Select-Project-Join (SPJ) transformations, logical provenance specifications encode minimal provenance. We have built a prototype system supporting the features and algorithms presented in the paper, and we report a few preliminary experimental results.**

I. Introduction

We address the problem of defining, generating, and tracing *provenance* in a setting of *data-oriented workflows*. In the workflow setting that we consider, *input data sets* are batch-processed through a graph of *transformations*, producing *intermediate data sets* and finally *output data sets*. In general, provenance retains information about how data propagates through the workflow.

Although existing systems such as [24], [27] are useful for building and executing workflows, capturing and exploiting provenance enables additional useful functionality in a workflow setting. For example, since provenance captures where data came from and how it is processed through the workflow, it can be very useful in supporting *debugging* and *drill-down*.

- **Debugging:** Workflows can have errors, either in the input data or the transformations. Finding such errors can be difficult and time-consuming—it may involve manually stepping through portions of the workflow on subsets of the data.
- **Drill-down:** Given a particular output element (or subset), a deeper understanding of how that element was generated may yield additional insights. Finding the relevant input data and processing steps for a specific output element is a feature not provided in most workflow systems.

There has been a great deal of past work on provenance in workflows. However, most approaches either track *coarse-grained* (schema and/or transformation level) provenance (e.g., [7], [19]), which is insufficient to support element-level debugging and drill-down, or they track the provenance of individual data elements with *physical provenance*. Physical provenance requires each output element to be annotated with some type of identifier for the contributing input elements (e.g., [9], [26]).

We support debugging and drill-down using *logical provenance*—provenance information stored at the transformation level. For many transformations, including a large subset of SQL, logical provenance can be derived automatically from the transformation's specification, and it captures exactly the same information as physical provenance but in a much more compact fashion (one specification per transformation rather than one per data element). Furthermore, for these transformations, logical provenance incurs no overhead at workflow-execution time, whereas physical provenance requires at a minimum the capture and storage of pointers. For some transformations, we must "augment" the output to support logical provenance. In the worst case, the augmenting process unavoidably degenerates to the equivalent of storing physical provenance.

The contributions of this paper are as follows:

- **Foundations of provenance** (Section II). We give a new general definition of provenance, introducing the notions of *correctness*, *precision*, and *minimality*. In contrast to previous principled definitions of provenance (e.g., *minimal witnesses* [10]), our provenance definitions are not restricted to specific types of transformations.
- **Theoretical properties of workflow provenance** (Section III). Given a workflow with provenance defined for each of its transformations, we identify when provenance properties such as correctness and minimality carry over from individual transformations to the workflow as a whole.
- **Logical provenance specifications for transformations** (Section IV). We describe a simple logical-provenance specification language consisting of *attribute mappings* and *filters*.
- **Algorithm for provenance tracing** (Section V). We provide an algorithm for provenance tracing in workflows where logical provenance for each transformation is specified using our language. In our algorithm we perform provenance tracing at the schema level to the extent possible, although eventually accessing the data obviously is required.

*This work was supported by the National Science Foundation (IIS-0904497), the Boeing Corporation, and a KAUST research grant.

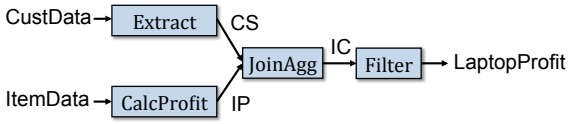

Fig. 1: Profit calculation workflow.

CustData

cust-id	country	activity_log	
C1	France	*<free text>*	(1)
C2	Germany	*<free text>*	(2)
C3	France	*<free text>*	(3)

CustSales (CS)

cust-id	country	item-id	quantity	
C1	France	I1	5	(1)
C1	France	I3	7	(2)
C2	Germany	I1	6	(3)
C2	Germany	I2	4	(4)
C3	France	I3	8	(5)

ItemData

item-id	brand	type	price	supplier_info	
I1	HP	laptop	700	*<free text>*	(1)
I2	Sony	tablet	550	*<free text>*	(2)
I3	Sony	laptop	800	*<free text>*	(3)

ItemProfit (IP)

item-id	brand	type	profit_per_item	
I1	HP	laptop	120	(1)
I2	Sony	tablet	200	(2)
I3	Sony	laptop	10	(3)

ItemCountryProfit (IC)

item-id	country	brand	type	profit	
I1	France	HP	laptop	600	(1)
I1	Germany	HP	laptop	720	(2)
I2	Germany	Sony	tablet	800	(3)
I3	France	Sony	laptop	150	(4)

LaptopProfit (LP)

item-id	country	brand	profit	
I1	France	HP	600	(1)
I1	Germany	HP	720	(2)
I3	France	Sony	150	(3)

Fig. 2: Profit calculation workflow, sample data.

- **Logical provenance for relational transformations** (Section VI). We consider logical provenance in the relational setting, observing that for a class of *Select-Project-Join* (SPJ) *transformations*, logical provenance specifications encode minimal provenance.
- An **implementation** of the logical-provenance approach as part of the *Panda* (for *Provenance And Data*) system at Stanford (Section VII). Panda permits arbitrary data-oriented workflows with each transformation specified in either SQL or in Python. We describe how Panda generates logical provenance specifications and executes our tracing algorithm. We also present some performance results (Section VIII).

The remainder of this section presents a running example. In Section IX we survey related work, and in Section X we conclude and propose future directions.

A. Running Example

Our example is obviously unrealistic, crafted to demonstrate the major features of our approach in a compact fashion. We represent the data sets in our example as simple tables, although in general our approach only requires that each data set is a collection of *elements*, and there are some designated *attributes* that are present in each data element. (The remainder of each element can be structured arbitrarily.)

Consider "WebShop," an online reseller. WebShop periodically runs a workflow, shown in Figure 1, to calculate the total profits generated from sales for items that WebShop sells. The workflow's input data sets are:

- CustData(cust-id, country, activity_log), where activity_log is a free-text field containing customer activity such as clicks and purchases.
- ItemData(item-id, brand, type, price, supplier_info), where supplier_info is a free-text field containing information, including cost, from the item's supplier.

The workflow involves the following transformations:

- Transformation **Extract** extracts from the activity_log attribute in CustData the items that each customer purchased, producing table CustSales(cust-id, country, item-id, quantity), abbreviated CS.
- Transformation **CalcProfit** calculates from the price and supplier_info attributes, for each item in ItemData, the profit made per item sold, producing table ItemProfit(item-id, brand, type, profit_per_item), abbreviated IP.
- Transformation **JoinAgg** joins tables CustSales and ItemProfit on attribute item-id, aggregating for each (item-id, country) pair the total profit from the item's sales in that country, producing table ItemCountryProfit(item-id, country, brand, type, profit), abbreviated IC.
- Transformation **Filter** filters ItemCountryProfit, selecting tuples with type equal to 'laptop', producing table LaptopProfit(item-id, country, brand, profit), abbreviated LP.[1]

Figure 2 shows sample input data sets along with all intermediate data, and finally output table LaptopProfit.

As a simple example of how provenance tracing can be useful for debugging, suppose we wanted to see why French profits on laptop sales with item-id I3 were so low. By tracing the provenance of LaptopProfit element (3) back through the workflow to ItemData, we discover that the profit per laptop (10, which may seem low) was extracted from the free-text field in ItemData element (3), suggesting that either the free-text field contains erroneous data or that the transformation **CalcProfit** has a bug.

[1]We assume intermediate data set ItemCountryProfit may be processed by other transformations as well; otherwise the workflow would of course filter for laptops much earlier.

II. Foundations of Provenance

Let a *data set* be any set of data *elements*. We assume each data set has some predefined *attributes* that are present in each element. Elements may contain arbitrary additional data, which we do not use for provenance. Attributes will not be used until Section IV, when we present our language for specifying logical provenance. Our formal definitions for provenance in this section and Section III are independent of element structure.

A *transformation* is any procedure that takes one or more data sets as input and produces one or more data sets as output. Recall from the running example that transformations are not always relational queries. A *workflow* is a directed acyclic graph, where nodes denote transformations, and edges denote the flow of the data sets that are input to and output from the transformations.

In much of this paper we consider transformations with one input set and one output set, for clarity of presentation. In an extended technical report [21], we show how all of our results generalize naturally to multiple input and output sets. Let $O = T(I)$ be a transformation instance, denoting the application of transformation T to input set I, obtaining output set O. (We will assemble transformations into workflows in Section III.) We assume $T(\emptyset) = \emptyset$. Our goal is to specify data-level provenance relationships between output elements in O and input elements in I. Specifically, given an output element $o \in O$, we would like to know the input subset $I^* \subseteq I$ that "produced" o. Defining this notion of provenance formally for general transformations can be challenging, as evidenced by a variety of definitions in the literature [12], [14], [15].

We give a new general definition of provenance for general transformations, introducing the notions of *correctness*, *precision*, and *minimality*. Intuitively, correct provenance $I^* \subseteq I$ must contain the "essence" of what derives o. Correct provenance typically is not unique, so we say correct provenance I^* is more precise than correct provenance I^{**} if $I^* \subset I^{**}$. It turns out with our definitions that there always exists a most precise provenance, which we refer to as the minimal provenance.

A. Correctness

Recall that correct provenance for an output element $o \in O$ should contain the "essence" of what derives o. Here is our formal definition:

Definition 2.1 (Correctness): Let $O = T(I)$ be a transformation instance. Consider an output element $o \in O$. Provenance $I^* \subseteq I$ is *correct* for o with respect to T if:

- For any $I' \subseteq I$, $o \in T(I')$ if and only if $o \in T(I' \cap I^*)$. □

Intuitively, this definition says that I^* is correct provenance if given any input subset I', we only need to consider $T(I' \cap I^*)$ to determine whether or not $o \in T(I')$. In other words, the only input elements in I' that could contribute to o are those that are also in I^*. Note that setting $I' = I$ shows that if I^* is correct for o, then $o \in T(I^*)$.

Example 2.1: Let $o = \mathsf{CustSales}(1)$, i.e., the first element from data set CustSales in Figure 2. Let $I^* = \{\mathsf{CustData}(1), \mathsf{CustData}(3)\}$. I^* is correct provenance although we will soon see that it is not minimal. Intuitively, we would expect I^* to be correct for o, since I^* contains the one element CustData(1) that derives o. To check formally that I^* is correct for o with respect to transformation **Extract**, we can simply verify that for each of the eight subsets $I' \subseteq \mathsf{CustData}$, the condition in Definition 2.1 is satisfied. □

B. Precision

Correctness alone does not capture the intuitive notion of provenance. For example, setting I^* equal to the entire input set I always yields correct provenance, but obviously this choice of I^* rarely gives useful information. Given two different subsets of I that are both correct provenance for an output element o, we prefer the smaller subset, since the smaller subset tells us more about what actually contributed to o.

Definition 2.2 (Precision): Let $O = T(I)$ and let $o \in O$. Suppose subsets $I^* \subseteq I$ and $I^{**} \subseteq I$ are both correct provenance for o with respect to T. I^* is *at least as precise* as I^{**} if $I^* \subseteq I^{**}$. I^* is *more precise* than I^{**} if $I^* \subset I^{**}$. □

Example 2.2: Consider again $o = \mathsf{CustSales}(1)$. Let $I^* = \{\mathsf{CustData}(1)\}$. It is easy to verify that I^* is correct provenance for o with respect to transformation **Extract**. Furthermore, since $I^* \subset \{\mathsf{CustData}(1), \mathsf{CustData}(3)\}$, I^* is more precise than the provenance from Example 2.1. □

C. Minimality

It turns out that there always exists a single most precise provenance, which we refer to as the minimal provenance. Intuitively, the minimal provenance captures "the essence" of what derives o.

Definition 2.3 (Minimality): Let $O = T(I)$ and let $o \in O$. Suppose subset $I^* \subseteq I$ is correct provenance for o with respect to T. I^* is *minimal* for o with respect to T if there does not exist provenance I^{**} that is correct and more precise than I^*. □

Theorem 2.1 (Unique Minimal Provenance): Let $O = T(I)$ and let $o \in O$. Let $I_1^*, \ldots, I_n^*$ be all of the correct provenances for o with respect to T. Let $I^M = I_1^* \cap \ldots \cap I_n^*$. I^M is correct and minimal, and there is no other correct and minimal provenance for o with respect to T. □

Proofs are omitted from this submission due to space constraints. All proofs appear in our extended technical report [21].

We now discuss how minimal provenance compares to the related notion of *minimal witnesses* from [10].

Definition 2.4 (Minimal Witness): Let $O = T(I)$ be a transformation instance and let $o \in O$. Subset $I^w \subseteq I$ is a *minimal witness* of o with respect to T if: (1) $o \in T(I^w)$; and (2) for any proper subset $I' \subset I^w$, $o \notin T(I')$. □

As an example to illustrate the relationship between minimal witnesses and minimal provenance, consider a transformation that eliminates duplicates. For any output element o, each of the corresponding duplicates o in the input data set is a separate minimal witness, while the minimal provenance is the entire set of o's in the input. The following theorem generalizes this example, showing that for any output element o from a monotonic transformation, the union of all minimal witnesses of o is equal to o's minimal provenance.

Theorem 2.2: Let $O = T(I)$ be an instance of a monotonic transformation, and let $o \in O$. Let $I_1^w, \ldots, I_m^w$ be all of the minimal witnesses of o with respect to T, and let I^* be the minimal provenance of o with respect to T. Then $I^* = I_1^w \cup \ldots \cup I_m^w$. □

For nonmonotonic transformations, minimal provenance may be larger than the union of all minimal witnesses, i.e., there may be input elements other than those in the minimal witnesses that contributed to o. Thus, minimal provenance can be thought of as a generalization of minimal witnesses for nonmonotonic transformations.

III. Workflow Provenance

We discuss the theoretical properties of workflow provenance. We first give some formal definitions. We then identify when correctness and minimality carry over from individual transformations to the workflow as a whole.

Consider any transformations T_1 and T_2. The *composition* $T_1 \circ T_2$ of the two transformations is a transformation that first applies T_1 to an input data set I_1 to obtain *intermediate* data set I_2. It then applies T_2 to I_2 to obtain output data set O.[2] Composition is associative, so we denote the linear composition of n transformations as $T_1 \circ T_2 \circ \cdots \circ T_n$. We refer to such a composition as a *data-oriented workflow*, or *workflow* for short.

Consider a workflow $T_1 \circ T_2 \circ \ldots \circ T_n$. A *workflow instance* is the application of the workflow to an input I_1. Let $I_{i+1} = T_i(I_i)$ for $i = 1..n$. The final output data set is I_{n+1}. We denote this workflow instance as $(T_1 \circ T_2 \circ \ldots \circ T_n)(I_1) = I_{n+1}$.

Suppose each of the transformations T_i includes a provenance specification. For each element $o \in I_{i+1}$, let $P_{T_i}(o) \subseteq I_i$ denote the provenance of o with respect to T_i. Then we can define workflow provenance in the intuitive recursive way as follows.

Definition 3.1 (Workflow Provenance): Let $W = T_1 \circ T_2 \circ \ldots \circ T_n$. Consider a workflow instance $(T_1 \circ T_2 \circ \ldots \circ T_n)(I_1) = I_{n+1}$, with initial input I_1. Let e be any data element involved in the workflow—input, intermediate, or output. The *workflow provenance* of e in W, denoted $P_W(e)$, is an input subset $I_1^* \subseteq I_1$. If e is an initial input element, i.e., $e \in I_1$, then $P_W(e) = \{e\}$. Otherwise, let T be the transformation that output e. Let $P_T(e)$ be the one-level provenance of e with respect to T. Then $P_W(e) = \bigcup_{e' \in P_T(e)} P_W(e')$. □

[2]Recall again that we consider single-input single-output transformations for presentation purposes in this paper, but generalize in our extended technical report [21].

Having defined workflow provenance in the intuitive way, we are interested in determining when properties such as correctness and minimality carry over from the individual transformations' provenance to the workflow provenance. We first define some transformation properties.

Definition 3.2 (Monotonicity): A transformation T is *monotonic* if for any input sets I and I', if $I \subseteq I'$, then $T(I) \subseteq T(I')$. □

Monotonicity is a common property, but is not always exhibited. In our running example of Section I-A, all of the transformations except **JoinAgg** (technically requiring the multi-input formalism in our extended technical report [21]) are monotonic.

Definition 3.3 (1-m Transformations): A transformation T is *1-m* if for any input set I, each output element $o \in T(I)$ has exactly one input element in its minimal provenance. □

Definition 3.4 (m-1 Transformations): A transformation T is *m-1* if for any input set I, each input element $e \in I$ is in at most one output element's minimal provenance. □

Let us now introduce a provenance property weaker than correctness, which we call *weak correctness*. We will see cases where in a workflow we can guarantee weak correctness but not correctness.

Definition 3.5 (Weak Correctness Definition): Let $O = T(I)$ be a transformation instance and let $o \in O$. We say that provenance $I^* \subseteq I$ is *weakly correct* for o with respect to T if:

- For any $I' \subseteq I$, if $I^* \subseteq I' \subseteq I$, then $o \in T(I')$. □

This definition states that I^* is weakly correct provenance for o with respect to T if T produces o when given any superset of I^* as input. Note that for any monotonic transformation T, if $o \in T(I^*)$, then I^* is weakly correct for o with respect to T.

As a simple example of how weak correctness and correctness differ, consider an instance of a transformation that performs "deduplication" (merging of elements deemed to represent the same real-world entity). Given an output element, any of the corresponding duplicates from the input set would alone constitute weakly correct provenance. However, correct provenance must contain all of the duplicates.

A. Preservation of Correctness

If we have correct provenance for each transformation in the workflow, and each transformation is monotonic, then the workflow provenance (as defined in Definition 3.1) is also correct.

Theorem 3.1: Let $W = T_1 \circ T_2 \circ \ldots \circ T_n$. Consider a workflow instance $(T_1 \circ T_2 \circ \ldots \circ T_n)(I_1) = I_{n+1}$ in which all of the transformations are monotonic. For each T_i and each element $e \in I_{i+1}$, let $P_{T_i}(e) \subseteq I_i$ be correct provenance for e with respect to T_i. Consider any output element $o \in I_{n+1}$. Workflow provenance $P_W(o)$ is correct for o with respect to W. □

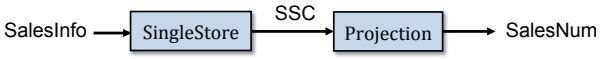

Fig. 3: Counterexample for correct workflow provenance.

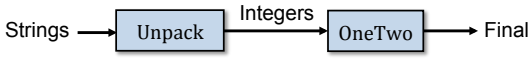

Fig. 4: Counterexample for minimal workflow provenance.

Monotonicity is key to our theorem: If we drop the assumption that all transformations are monotonic, then we have examples demonstrating that workflow provenance is not necessarily correct.

Example 3.1: Consider workflow $W = \mathbf{SingleStore} \circ \mathbf{Projection}$ shown in Figure 3. The initial input data set SalesInfo contains country-sales pairs for each of a corporation's worldwide stores: $\mathsf{SalesInfo}(\mathsf{country}, \mathsf{sales}) = \{(France, 10), (Germany, 20), (Germany, 10)\}$. Transformation **SingleStore** retains sales information for stores in countries with only one store, producing intermediate data set SingleStoreCountries (abbreviated SSC): $\mathsf{SSC} = \mathbf{SingleStore}(\mathsf{SalesInfo}) = \{(France, 10)\}$. The following provenance is correct: $P_{\mathbf{SingleStore}}((France, 10)) = \{(France, 10)\} \subseteq \mathsf{SalesInfo}$. Transformation **Projection** projects away the country name, leaving only total sales for each of the stores in SSC: $\mathbf{Projection}(\mathsf{SSC}) = \{10\}$. Note that **SingleStore** is not monotonic, but **Projection** is. The following provenance is correct: $P_{\mathbf{Projection}}(10) = \{(France, 10)\}$. Let $o = 10 \in \mathsf{SalesNum}$. The workflow provenance $P_W(o)$ of o following Definition 3.1 is $\{(France, 10)\} \subseteq \mathsf{SalesInfo}$. However, the only correct provenance for o with respect to W is actually all of SalesInfo: $\{(France, 10), (Germany, 20), (Germany, 10)\}$. To see, for example, that $(Germany, 10)$ needs to be in correct provenance I^*, note that by setting $\mathsf{SalesInfo}' = \{(Germany, 10)\}$, we get $o \in W(\mathsf{SalesInfo}')$. However, if $(Germany, 10) \notin I^*$, then $o \notin W(\mathsf{SalesInfo}' \cap I^*)$. Thus, workflow provenance $P_W(o)$ is not correct. □

B. Preservation of Minimality

Although correctness carries over from the provenance of individual transformations to workflow provenance when all transformations are monotonic, once again, monotonicity is key. We now show an example demonstrating that although correctness carries over, there does not exist such a guarantee for minimality.

Example 3.2: Consider workflow $W = \mathbf{Unpack} \circ \mathbf{OneTwo}$ shown in Figure 4. The input data set Strings contains two strings: {"1", "1, 2"}. Transformation **Unpack** finds all integers contained in Strings and removes duplicates, producing intermediate data set $\mathsf{Integers} = \{1, 2\}$. We have minimal provenance for **Unpack**: $P_{\mathbf{Unpack}}(1) = \{\text{"1"}, \text{"1, 2"}\}$ and $P_{\mathbf{Unpack}}(2) = \{\text{"1, 2"}\}$. Transformation **OneTwo** processes intermediate data set Integers as follows:

- If $1 \in \mathsf{Integers}$ and $2 \in \mathsf{Integers}$, then $\mathbf{OneTwo}(\mathsf{Integers}) = \{\text{"success"}\}$
- Else, $\mathbf{OneTwo}(\mathsf{Integers}) = \emptyset$

In this workflow instance, $\mathsf{Final} = \mathbf{OneTwo}(\mathsf{Integers}) = \{\text{"success"}\}$. The minimal provenance of "success" with respect to **OneTwo** is $P_{\mathbf{OneTwo}}(\text{"success"}) = \{1, 2\}$.

Note that **Unpack** and **OneTwo** are both monotonic. Let $o =$ "success" $\in$ Final. The workflow provenance $P_W(o)$ of o according to Definition 3.1 is {"1", "1, 2"}. However, {"1, 2"} is also correct provenance for o with respect to W, and it is minimal. Thus, workflow provenance in this example is not minimal. □

Intuitively, in the above example, workflow provenance is not minimal because it contains an input element e ("1" in our example) that can never actually influence whether the output element is produced. In other words, the uninfluential input element e's contribution is never sufficient to change the ultimate outcome, yet e is included in the provenance.

Let us identify a class of workflows for which the above situation does not occur. If a monotonic workflow consists of a set of m-1 transformations followed by a set of 1-m transformations, and we have minimal provenance for each transformation in the workflow, then workflow provenance is minimal.

Theorem 3.2: Let $W = T_1 \circ T_2 \circ \ldots \circ T_n$. Consider a workflow instance $(T_1 \circ T_2 \circ \ldots \circ T_n)(I_1) = I_{n+1}$ in which all of the transformations are monotonic, $T_1, \ldots, T_j$ are m-1 transformations, and $T_{j+1}, \ldots, T_n$ are 1-m transformations. For each T_i and each element $e \in I_{i+1}$, let $P_{T_i}(e) \subseteq I_i$ be minimal provenance for e with respect to T_i. Consider any output element $o \in I_{n+1}$. Workflow provenance $P_W(o)$ is minimal for o with respect to W. □

C. Preservation of Weak Correctness

Given a workflow with at most one nonmonotonic transformation, if we have weakly correct provenance at each transformation, then workflow provenance is guaranteed to be weakly correct.

Theorem 3.3: Let $W = T_1 \circ T_2 \circ \ldots \circ T_n$. Consider a workflow instance $(T_1 \circ T_2 \circ \ldots \circ T_n)(I_1) = I_{n+1}$ in which at most one of the transformations is nonmonotonic. For each T_i and each element $e \in I_{i+1}$, let $P_{T_i}(e) \subseteq I_i$ be weakly correct provenance for e with respect to T_i. Consider any output element $o \in I_{n+1}$. Workflow provenance $P_W(o)$ is weakly correct for o with respect to W. □

Given a workflow with two nonmonotonic transformations, workflow provenance is not guaranteed to be correct or weak correct.

Example 3.3: Consider workflow $W = \mathbf{OneTwo} \circ \mathbf{ThreeFour}$ shown in Figure 5. Let $\mathsf{Initial} = \{1, 2\}$. Transformation **OneTwo** produces intermediate data set Int as follows:

- If $1 \in \mathsf{Initial}$ and $2 \in \mathsf{Initial}$, then $\mathbf{OneTwo}(\mathsf{Initial}) = \{3\}$
- Else if $1 \in \mathsf{Initial}$ and $2 \notin \mathsf{Initial}$, then $\mathbf{OneTwo}(\mathsf{Initial}) = \{3, 4\}$
- Else, $\mathbf{OneTwo}(\mathsf{Initial}) = \emptyset$

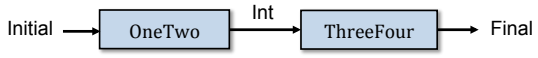

Fig. 5: Counterexample for weakly correct workflow provenance.

In this workflow instance, $\mathsf{Int} = \{3\}$. The following is correct (and therefore weakly correct) provenance for 3 with respect to **OneTwo**: $P_{\mathbf{OneTwo}}(3) = \{1\}$. (Intuitively $\{1\}$ is correct, and in fact minimal, because any time Initial contains 1, Int contains 3.) Transformation **ThreeFour** produces Final from Int as follows:

- If $3 \in \mathsf{Int}$ and $4 \notin \mathsf{Int}$, then $\mathbf{ThreeFour}(\mathsf{Int}) = \{5\}$
- Else, $\mathbf{ThreeFour}(\mathsf{Int}) = \emptyset$

In this workflow instance, $\mathsf{Final} = \{5\}$. The following provenance is correct (and therefore weakly correct) for 5 with respect to **ThreeFour**: $P_{\mathbf{ThreeFour}}(5) = \{3\}$. Neither **OneTwo** nor **ThreeFour** is monotonic. Let $o = 5 \in \mathsf{Final}$. The workflow provenance $P_W(o)$ of o is $\{1\}$. However, the only weakly correct provenance of o with respect to W is $\{1, 2\}$, since $5 \notin W(\{1\}) = \emptyset$. Thus, workflow provenance is not weakly correct. □

IV. Logical Provenance Specifications

Having discussed the desirable formal properties of provenance in Section II and how these properties carry over from individual transformations to a workflow in Section III, we now discuss how to actually specify provenance relationships between the input elements and output elements of a given transformation. Recall we assume each data set has some predefined attributes that are present in each element. Elements may contain arbitrary additional data, which we do not use for provenance.

We describe a simple logical-provenance (transformation level) specification language. This language can encode precise (data-element level) provenance for a large class of transformations, capturing the same information as physical provenance but in a much more compact fashion. Logical provenance specifications in our language consist of *attribute mappings* and *filters*.

Definition 4.1 (Attribute Mapping): Let T be a transformation with input set I and output set O. Let A be an attribute of input set I, and let B be an attribute of output set O. T has an *attribute mapping* between input attribute $I.A$ and output attribute $O.B$, denoted $I.A \leftrightarrow O.B$, if for all possible values of $x \in domain(B)$ and all possible input set instances I':
$\sigma_{B=x}(T(I')) = \sigma_{B=x}(T(\sigma_{A=x}(I')))$ □

In words, attribute mapping $I.A \leftrightarrow O.B$ states that the output subset of O where $B = x$ is unaffected by all elements in I except those where $A = x$. Attribute mappings can also involve functions, e.g., $f(I.A) \leftrightarrow g(O.B)$, but we do not elaborate on that extension here.

Example 4.1: Consider transformation **JoinAgg** from the running example (Figure 1), which takes data sets CustSales (CS) and ItemProfit (IP) as input and joins them on attribute item-id to output data set ItemCountryProfit (IC). Attribute mapping IP.brand $\leftrightarrow$ IC.brand is one example that holds for this transformation, indicating that the subset of output elements in IC with particular brand values is produced by the input subset of IP with the same brand values. (Technically this example needs our extended formalism for multiple input sets, but it should be clear nonetheless.) Other attribute mappings that hold for transformation **JoinAgg** are IP.item-id $\leftrightarrow$ IC.item-id, IP.type $\leftrightarrow$ IC.type, CS.country $\leftrightarrow$ IC.country, and CS.item-id $\leftrightarrow$ IC.item-id. □

Definition 4.2 (Filter): Let T be a transformation with input set I and output set O. T has a *filter condition* C on input I if for all possible input set instances I', $T(I') = T(\sigma_C(I'))$. □

In words, filter C states that output elements are never affected by input elements from I that do not satisfy condition C.

Example 4.2: Consider transformation **Filter** from the running example, which takes data set ItemCountryProfit (IC) as input and outputs data set LaptopProfit (LP). The filter type='laptop' holds for this transformation, indicating that output elements in LP are never affected by elements in IC except those satisfying type='laptop'. □

We now define provenance as encoded by attribute mappings and filters. We will shortly present a theorem asserting its correctness.

Definition 4.3 (Logically Encoded Provenance): Consider a transformation instance $O = T(I)$ with a logical provenance specification consisting of a set of attribute mappings M and a set of filters F. We denote the specification as (M, F). Consider an output element $o \in O$. The provenance of o as encoded by (M, F) is $I^* = \sigma_{C^*}(I)$ where $C^* = (\bigwedge_{(I.A \leftrightarrow O.B) \in M} A = o.B) \wedge (\bigwedge_{C \in F} C)$. □

Example 4.3: Consider again transformation **Filter** from the running example, which takes data set ItemCountryProfit (IC) as input and outputs data set LaptopProfit (LP). Consider the logical provenance specification (M, F), where $M = \{$IC.item-id$\leftrightarrow$LP.item-id, IC.country $\leftrightarrow$ LP.country, IC.brand $\leftrightarrow$ LP.brand$\}$ and $F = \{$type='laptop'$\}$. Referring to Figure 2, if $o = \mathsf{LP}(1)$, the first tuple in LP, then o's provenance as encoded by (M, F) is $\sigma_{\text{item-id='I1'} \wedge \text{country='France'} \wedge \text{brand='HP'} \wedge \text{type='laptop'}}(\mathsf{IC}) = \{\mathsf{IC}(1)\}$. □

The following theorem states that provenance encoded by logical specifications is indeed correct.

Theorem 4.1: Consider a transformation instance $O = T(I)$ with logical provenance specification (M, F). Given any output element $o \in O$, o's provenance $I^* = \sigma_{C^*}(I)$ as specified in Definition 4.3 is correct for o with respect to T. □

A. Multi-Attribute Mappings

Given attribute mappings involving single input and output attributes, we can combine them to create attribute mappings across sets of attributes.

Definition 4.4 (Multi-Attribute Mapping): Let T be a transformation with input set I and output set O. Let $\mathbf{A} = (A_1, A_2, \ldots, A_n)$ and $\mathbf{B} = (B_1, B_2, \ldots, B_n)$ denote vectors of attributes from I and O respectively. Let $\sigma_{\mathbf{A}=\mathbf{x}}$ be shorthand for $\sigma_{A_1=x_1 \wedge \ldots \wedge A_n=x_n}$, and let $\sigma_{\mathbf{B}=\mathbf{x}}$ be shorthand for $\sigma_{B_1=x_1 \wedge \ldots \wedge B_n=x_n}$. Attribute mapping $I.\mathbf{A} \leftrightarrow O.\mathbf{B}$ holds if for all possible values of $\mathbf{x} \in domain(\mathbf{B})$ and all possible input set instances I':

$\sigma_{\mathbf{B}=\mathbf{x}}(T(I')) = \sigma_{\mathbf{B}=\mathbf{x}}(T(\sigma_{\mathbf{A}=\mathbf{x}}(I')))$ □

Example 4.4: Consider again transformation **JoinAgg** from the running example. Attribute mapping (IP.item-id, IP.brand, IP.type) ↔ (IC.item-id, IC.brand, IC.type) holds for this transformation, indicating that the subset of output elements in IC with particular (item-id, brand, type) values corresponds to the input subset of IP with the same (item-id, brand, type) values. □

The following theorem states that a set of single-attribute mappings is equivalent to its combination:

Theorem 4.2 (Combining Attribute Mappings): Let T be a transformation with input set I and output set O. Let $\mathbf{A} = (A_1, A_2, \ldots, A_n)$ and $\mathbf{B} = (B_1, B_2, \ldots, B_n)$ denote vectors of attributes from I and O respectively. Attribute mappings $I.A_1 \leftrightarrow O.B_1, \ldots, I.A_n \leftrightarrow O.B_n$ hold if and only if $I.\mathbf{A} \leftrightarrow O.\mathbf{B}$ holds. □

V. Provenance Tracing in Workflows

Consider a workflow in which each transformation has a logical provenance specification in the language described in Section IV. Given an output element in the workflow, suppose we want to find the input subsets that contributed to the output element, e.g., for debugging or drill-down purposes. In this section, we discuss how to perform *provenance tracing*, i.e., how to use the logical provenance specifications given for each transformation in the workflow to compute the workflow provenance of a given output element. In particular, we are interested in making provenance tracing efficient.

As we saw in Section III, even if we have minimal provenance for each transformation, workflow provenance isn't always minimal, or even correct. Using the theoretical results of Section III, for many workflows we can guarantee the minimality or correctness of workflow provenance. For workflows where minimality or correctness of workflow provenance is not theoretically guaranteed, we would still like to provide the option of computing workflow provenance, since it could still be helpful for debugging and drill-down.

Although it is straightforward to trace provenance recursively backwards through a workflow one transformation at a time, sometimes we can combine logical provenance across transformations, enabling more efficient tracing. In Section V-A, we specify when and how logical provenance can be combined across transformations without losing correctness. In Section V-B, we give conditions under which tracing the combined logical provenance is guaranteed to return the same result as tracing each transformation's logical specification separately (thus, e.g., preserving minimality). In Section V-C, we present our overall algorithm for provenance tracing in workflows with logical provenance specifications given for each transformation.

A. Combining Logical Provenance Across Transformations

Let T_1 and T_2 be transformations such that T_2 takes as input T_1's output data set: $T_1(I_1) = I_2$ and $T_2(I_2) = I_3$. Given an element o from output set I_3, suppose we wanted to trace o's provenance to I_1 using the logical provenance specifications for T_1 and T_2. We specify when and how logical provenance can be combined across T_1 and T_2 without losing correctness. Throughout this section, we use $W = T_1 \circ T_2$ to denote the workflow composed of T_1 and T_2, with input set I_1, intermediate set I_2, and output set I_3. Let T_1 have logical provenance specification (M_1, F_1), and let T_2 have logical provenance specification (M_2, F_2). We first show how attribute mappings can be combined across transformations.

Theorem 5.1 (Transitivity of Attribute Mappings): Consider $W = T_1 \circ T_2$. If T_1 has attribute mapping $(I_1.A \leftrightarrow I_2.B) \in M_1$, and T_2 has attribute mapping $(I_2.B \leftrightarrow I_3.C) \in M_2$, then attribute mapping $(I_1.A \leftrightarrow I_3.C) \in M_W$ holds for W. □

Example 5.1: Consider transformations **JoinAgg** and **Filter** from the running example (Figure 1). Since attribute mapping IP.brand ↔ IC.brand holds for **JoinAgg**, and attribute mapping IC.brand ↔ LP.brand holds for **Filter**, then by Theorem 5.1 attribute mapping IP.brand ↔ LP.brand holds for $\mathbf{JoinAgg} \circ \mathbf{Filter}$. Informally, the composite mapping states that the subset of output elements in LaptopProfit with a particular brand value are derived from the subset of ItemProfit with the same brand value. □

Now consider filters. It is straightforward to show that filters that hold for T_1 also hold for $T_1 \circ T_2$.

Theorem 5.2: Consider $W = T_1 \circ T_2$. If filter condition $C \in F_1$ holds for T_1, then $C \in F_W$ also holds for W. □

The next theorem shows that in some cases filters on T_2 can be "propagated" backward through I_2 using attribute mappings so they hold on W.

Theorem 5.3: Consider $W = T_1 \circ T_2$. Suppose T_2 has a filter condition $C \in F_2$. If $(I_1.A_1 \leftrightarrow I_2.B_1, \ldots, I_1.A_s \leftrightarrow I_2.B_s) \subseteq M_1$ where $B_1, \ldots, B_s$ are all attributes that are involved in C, then filter condition $C' \in F_W$ holds for W, where C' is equal to C after replacing all B_i with A_i. □

B. Relationship Between Combined Logical Provenance and Workflow Provenance

Although attribute mappings and filters can be combined across transformations using Theorems 5.1, 5.2, and 5.3, sometimes the combined logical provenance is weaker (less precise) than the workflow provenance computed by considering transformations separately, accessing intermediate data between transformations.

Example 5.2: Consider workflow $W = \mathbf{MultiStore} \circ \mathbf{Projection}$ shown in Figure 6. The initial input data set

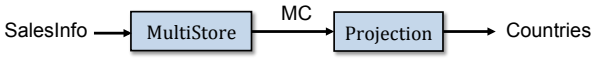

Fig. 6: Example where combining logical provenance is not equivalent to workflow provenance.

SalesInfo contains country-city-sales triples for each of a corporation's worldwide stores: SalesInfo(country, city, sales) = $\{(France, Paris, 10), (France, Paris, 20), (France, Nice, 30)\}$. Transformation **MultiStore** retains cities with more than one store, producing intermediate data set MultiCities (abbreviated MC): MC(country, city) = **MultiStore**(SalesInfo) = $\{(France, Paris)\}$. Attribute mappings SalesInfo.country $\leftrightarrow$ MC.country and SalesInfo.city $\leftrightarrow$ MC.city both hold for **MultiStore**. Transformation **Projection** projects away the city name, leaving only countries (with duplicates eliminated): Countries(country) = $\{France\}$. Attribute mapping MC.country $\leftrightarrow$ Countries.country holds for **Projection**. Let $o = France \in$ Countries. The workflow provenance $P_W(o)$ of o following Definition 3.1 is $\{(France, Paris, 10), (France, Paris, 20)\} \subseteq$ SalesInfo. By Theorem 5.1, attribute mapping SalesInfo.country $\leftrightarrow$ Countries.country holds for W. However, the provenance of $o = France \in$ Countries using this composite attribute mapping is $\{(France, Paris, 10), (France, Paris, 20), (France, Nice, 30)\}$, which is correct, but not as precise as the workflow provenance $\{(France, Paris, 10), (France, Paris, 20)\}$ we computed by keeping the attribute mappings separate. □

The following theorem states that we can combine logical provenance across transformations without losing the precision of o's workflow provenance if: (1) all attribute mappings $A \leftrightarrow B$ in M_1 have a corresponding attribute mapping $B \leftrightarrow D$ in M_2, and (2) the provenance of o encoded by (M_2, F_2) for I_2 is nonempty.

Theorem 5.4: Consider workflow $W = T_1 \circ T_2$. Suppose $M_1 = \{A_1 \leftrightarrow B_1, \ldots, A_r \leftrightarrow B_r\}$ and $M_2 = \{B_1 \leftrightarrow D_1, \ldots, B_s \leftrightarrow D_s\}, r \leq s$. Let composite $M_W = \{A_1 \leftrightarrow D_1, \ldots, A_r \leftrightarrow D_r\}$. Consider a workflow instance $I_3 = W(I_1)$ and any output element $o \in I_3$. Suppose the provenance of o encoded by (M_2, F_2) for I_2 is nonempty. Let I_1^* be o's provenance as encoded by (M_W, F_1) for W according to Definition 4.3. Let I_1^{**} be the workflow provenance of o according to Definition 3.1. Then $I_1^* = I_1^{**}$. □

C. Tracing Algorithm

In workflows with logical specifications at each transformation, a range of provenance-tracing algorithms are possible. A conservative algorithm, for example, may combine logical provenance across transformations only when it is certain that doing so will not reduce precision. Alternatively a more aggressive algorithm (at least from the performance perspective) may combine logical provenance even without such certainty, to avoid the overhead of tracing provenance through each step of intermediate data. The tracing algorithm presented here attempts to avoid losing precision by combining logical provenance only when the primary condition of Theorem 5.4 is satisfied: Every attribute mapping $A \leftrightarrow B$ in M_1 has a corresponding mapping $B \leftrightarrow D$ in M_2. However, since the algorithm does not also check the second condition of Theorem 5.4 (whether the provenance of output elements at intermediate data sets is nonempty), in rare cases the algorithm can reduce precision. One area of future work is to study the relationship between the precision and performance of tracing algorithms, perhaps identifying classes of workflows for which tracing performance can be improved greatly without losing much precision. Note we give here our general algorithm allowing multiple input and output sets.

Algorithm 5.1 (Provenance Tracing): Consider a workflow instance in which each transformation T has logical provenance specification (M^T, F^T). Let $I_1, \ldots, I_m$ be the initial input sets of the workflow, and let I be any intermediate or output data set. PT recursively traces the provenance of subset $E \subseteq I$ using attribute mappings M and filters F. The initial invocation of PT to trace output element $o \in O_j$ is $PT(\{o\} \subseteq O_j, \emptyset, \emptyset, O_j)$.

$PT(E \subseteq I, M, F, I')$:
  if I' is an initial input set I_j **then**:
    if $I = I'$ **then**:
      let $I_j^* = E$
      return $\langle I_1^*, \ldots, I_m^* \rangle$, where each $I_k^* = \emptyset$ for $k \neq j$
    else:
      suppose $M = \{A_1 \leftrightarrow B_1, \ldots, A_r \leftrightarrow B_r\}$
      let $I_j^* = \bigcup_{e \in E} \sigma_{(A_1 = e.B_1) \wedge \ldots \wedge (A_r = e.B_r) \wedge F}(I_j)$
      return $\langle I_1^*, \ldots, I_m^* \rangle$, where each $I_k^* = \emptyset$ for $k \neq j$
  else:
    let T be the transformation that output I',
      with input sets $I_1^T, \ldots, I_{m_T}^T$
    for i in $[1, m_T]$:
      let M_i be the subset of mappings $A \leftrightarrow B$ in M^T
        such that A is from I_i^T and B is from I'
      let F_i be the subset of filters in F^T that are on I_i^T
      if $I = I'$ **then** for $1 \leq i \leq m_T$:
        let $\langle I_1^i, \ldots, I_m^i \rangle = PT(E \subseteq I, M_i, F_i, I_i^T)$
      else:
        if every right attribute in M_i is a left attribute of M:
          suppose $M_i = \{A_1 \leftrightarrow B_1, \ldots, A_r \leftrightarrow B_r\}$
          suppose $M = \{B_1 \leftrightarrow D_1, \ldots, B_s \leftrightarrow D_s\}$, $r \leq s$
          let $M' = \{A_1 \leftrightarrow D_1, \ldots, A_r \leftrightarrow D_r\}$
          let $\langle I_1^i, \ldots, I_m^i \rangle = PT(E \subseteq I, M', F_i, I_i^T)$
        else:
          suppose $M = \{A_1 \leftrightarrow B_1, \ldots, A_r \leftrightarrow B_r\}$
          let $E' = \bigcup_{e \in E} \sigma_{(A_1 = e.B_1) \wedge \ldots \wedge (A_r = e.B_r) \wedge F}(I_i^T)$
          let $\langle I_1^i, \ldots, I_m^i \rangle = PT(E' \subseteq I_i^T, \emptyset, \emptyset, I_i^T)$
    return $\langle I_1^*, \ldots, I_m^* \rangle$, where each $I_j^* = \bigcup_{1 \leq i \leq m_T} I_j^i$

VI. Logical Provenance for Relational Transformations

We now consider logical provenance in the relational setting, discussing how to generate logical provenance specifications for *Select-Project-Join* (*SPJ*) *transformations*. (In

our extended technical report [21], we extend our results to *Select-Project-Join-Aggregate* (*SPJA*) *transformations*.) For a wide class of SPJ transformations, provenance encoded by our logical specifications is minimal. Note in this section we assume transformations with multiple input sets.

Definition 6.1 (Select-Project-Join Transformation): A *Select-Project-Join* (*SPJ*) *transformation* is any transformation that can be expressed as a tree of relational algebra *selection* (σ), *projection* (π), and *cross-product* ($\times$) *operators*. We assume set semantics, so π is duplicate-eliminating. □

All SPJ transformations T can be transformed into a canonical form $T(I_1, \ldots, I_m) = \pi_{\mathbf{A}}(\sigma_C(\sigma_{C_1}(I_1) \times \ldots \times \sigma_{C_m}(I_m)))$ using a sequence of algebraic transformations [30]. Note $\mathbf{A}$ is the final projection list, C_i are "local" (single-table) selection conditions, and C contains "join" (multi-table) conditions.

Given the canonical form of an SPJ transformation T, it is straightforward to generate a set of attribute mappings and filters that hold for T.

Theorem 6.1: Let T be an SPJ transformation: $O = T(I_1, \ldots, I_m) = \pi_{\mathbf{A}}(\sigma_C(\sigma_{C_1}(I_1) \times \ldots \times \sigma_{C_m}(I_m)))$. For each attribute $I_i.A \in \mathbf{A}$, attribute mapping $I_i.A \leftrightarrow O.A$ holds for T. □

Theorem 6.2: Let T be an SPJ transformation: $O = T(I_1, \ldots, I_m) = \pi_{\mathbf{A}}(\sigma_C(\sigma_{C_1}(I_1) \times \ldots \times \sigma_{C_m}(I_m)))$. For each I_i, T has filter condition C_i on I_i. □

Given the above theorems, we can generate a *canonical logical provenance specification* for any SPJ transformation.

Definition 6.2 (Canonical Logical Specification): Let T be an SPJ transformation: $O = T(I_1, \ldots, I_m) = \pi_{\mathbf{A}}(\sigma_C(\sigma_{C_1}(I_1) \times \ldots \times \sigma_{C_m}(I_m)))$. The *canonical logical provenance specification* for T is (M, F) where M contains all mappings $I_i.A \leftrightarrow O.A$ such that $I_i.A \in \mathbf{A}$, and F contains all C_i. □

The following theorem states that for all SPJ transformations in which the final output schema contains all attributes involved in the join conditions, the provenance encoded by the canonical logical specification is minimal.

Theorem 6.3: Consider an SPJ transformation instance: $O = T(I_1, \ldots, I_m) = \pi_{\mathbf{A}}(\sigma_C(\sigma_{C_1}(I_1) \times \ldots \times \sigma_{C_m}(I_m)))$. Suppose $\mathbf{A}$ contains all attributes in C. (Recall C contains all multi-table conditions.) Given $o \in O$, let $I_1^*, \ldots, I_m^*$ be the provenance of o as encoded by the canonical logical specifications for T. Each of $I_1^*, \ldots, I_m^*$ is minimal for o with respect to T. □

Given an SPJ transformation $T(I_1, \ldots, I_m) = \pi_{\mathbf{A}}(\sigma_C(\sigma_{C_1}(I_1) \times \ldots \times \sigma_{C_m}(I_m)))$, Theorem 6.3 only guarantees that the canonical logical specification given in Definition 6.2 encodes minimal provenance if $\mathbf{A}$ contains all attributes in C. One approach to solving this problem is to simply retain more attributes in $\mathbf{A}$ to meet the requirement, a process we call *augmentation*.

Definition 6.3 (Augmentation): Let T be a transformation with output schema $\mathbf{A}$. Transformation T' is an *augmentation* of transformation T if T' has output schema $\mathbf{A}' \supset \mathbf{A}$ and $T = T' \circ \pi_{\mathbf{A}}$. □

Augmentation can be applied to any transformation, including non-relational transformations. The benefit of augmentation is that it allows logical specifications to encode more precise provenance. However, augmentation requires more storage, and it also requires a logical or materialized view to be maintained, to produce the original output from the augmented one. Note that if augmentation retains all input attributes or even a key for each input tuple, then we effectively require each output tuple o to have a pointer to the input tuples that contributed to o. Thus, augmentation degenerates to physical provenance.

VII. System

We have built a prototype system called *Panda* (for *Provenance And Data*) that supports debugging and drill-down in workflows using either physical or logical provenance. We have previously described Panda in a demonstration description [20]. Here we focus on how Panda generates logical provenance specifications and executes our tracing algorithms.

Panda permits arbitrary data-oriented workflows with each transformation specified in either SQL or in Python. Currently, all data sets handled by Panda are encoded in relational tables, and all records (tuples) are given a globally unique ID. The main backend is **SQLite**, which stores all data sets, SQL transformations, provenance, and workflow information. For each transformation in a workflow, Panda generates logical provenance specifications consisting of attribute mappings and filters, as explained next.

A. SQL Transformations

Panda supports transformations specified as SQL queries, currently limited to single SELECT blocks with optional grouping and aggregation, with conjunctive conditions.

As an example, consider transformation **JoinAgg** from our running example (Figure 1). This transformation can be expressed using the following SQL query:

```
Create Table ItemCountryProfit As
Select CS.item-id, country, brand, type,
  SUM(quantity*profit_per_item) as profit,
From CustSales CS, ItemProfit IP
Where CS.item-id = IP.item-id
Group By CS.item-id, country, brand, type
```

Panda generates logical provenance specifications through syntactic analysis of SQL queries. During syntactic analysis, Panda generates attribute mappings between attributes appearing in the SELECT clause and all possible corresponding input attributes, computed by taking the transitive closure over equalities in the WHERE clause. Panda generates filters for all conjuncts in the WHERE clause that apply to a single input table. For our example transformation, Panda generates logical specification (M, F): $M = \{$CS.item-id $\leftrightarrow$ IC.item-id, CS.country $\leftrightarrow$ IC.country,

IP.item-id $\leftrightarrow$ IC.item-id, IP.brand $\leftrightarrow$ IC.brand, IP.type $\leftrightarrow$ IC.type$\}, F = \{\}$.

Now suppose the above query did not contain the item-id attribute in its SELECT clause. In this case Panda augments the query automatically to support logical provenance: Panda adds the item-id attribute to the SELECT clause for the purpose of provenance tracing, producing augmented result AugItemCountryProfit. It then creates a SQL view for the output data set to hide the extra attribute:

```
Create View ItemCountryProfit As
Select country, brand, type, profit,
From AugItemCountryProfit
```

Panda augments SPJ transformations (i.e., without aggregation) as described in Section VI: Transformations that do not satisfy the requirements of Theorem 6.3 are augmented so that logical provenance encodes minimal provenance. Further details and examples are given in our extended technical report [21].

B. Python Transformations

Currently, Panda only supports Python transformations that are one-one or one-many transformations—i.e., the transformation operates on one record at a time—although this restriction can be lifted with additional work. For Python transformations, Panda currently prefers for the user to specify logical provenance, but provides a fallback mechanism if no provenance is specified.

Consider transformation **Extract** from the running example. The user may provide Panda with the logical specification (M, F) for **Extract**: $M = \{$CustData.cust-id $\leftrightarrow$ CS.cust-id, CustData.country $\leftrightarrow$ CS.country$\}$, $F = \{\}$. If no user-specified logical provenance is provided, Panda augments each output record with the ID of the corresponding input record (calling it src_id in the output), and generates the attribute mapping I.ID $\leftrightarrow$ O.src_id for the augmented transformation. (ID is the column for the globally unique ID present in every Panda record.) As usual, Panda stores the output of the augmented transformation, and creates a view on the augmented output to produce the original output. Note that since each record in the augmented output effectively contains a pointer to its contributing input record, augmenting the output in this case degenerates to effectively storing physical provenance.

C. Provenance Tracing

Panda enables provenance to be traced along a specified *tracing path*: a *source data set* O, and a *target data set* I reached via a single specified path backwards from the source. To perform tracing, Panda generates and executes a SQL query that joins multiple tables, returning the same provenance as in Algorithm 5.1.

In more detail, given a record in O, Panda first combines logical provenance wherever possible along the tracing path from O to I. Then Panda generates a query that joins the data sets in the remaining combined path. The conditions in the join query are based on the logical provenance (attribute mappings and filters) for the data sets in the combined path, and on the unique ID of the tuple being traced. Further details and examples are given in our extended technical report [21].

VIII. Experiments

The primary goal of our experiments was to explore the benefits of logical provenance over physical provenance. We will see that logical provenance has smaller time and space overhead as expected (Section VIII-A), and also enables more efficient provenance tracing by combining provenance across transformations (Section VIII-B). Overhead and tracing performance may vary considerably depending on the particular workflow and data. Our experiments were not designed to explore these variations (planned for future work), but rather to serve as an example of how logical provenance compares to physical provenance in one fairly neutral setting.

We conducted our experiments using the Panda system on a workflow that processes health insurance claim data from the Heritage Health Prize competition [1]. The workflow's input data sets are:

- ClaimData(memberID, providerID, diseaseID, data), where attribute data is a text field containing health insurance claim data
- DiseaseData(diseaseID, type)
- VisitData(memberID, days), which contains the number of days spent in the hospital

All of our experiments were run on a MacBook Air laptop (1.8 GHz Intel Core i7, 4 GB memory, 250 GB storage, Mac OS X 10.7). To measure the performance of physical provenance, we used a modified version of Panda that stores physical provenance in separate provenance tables.

A. Time and Space Overhead

Figure 7 demonstrates the time overhead of logical and physical provenance, running our workflow with varying input data sizes. Comparing the running times of workflow computation with and without provenance, the time overhead is roughly proportional to the size of the input data set, ranging from 3% to 6% for logical provenance, and from 45% to 62% for physical provenance. The time overhead for logical provenance is due to the additional output data produced by augmentation.

Figure 8 demonstrates the space overhead of logical and physical provenance. The space measurement totals all intermediate and output data involved in the workflow. Logical provenance incurs a consistent approximately 4% space overhead across input data sizes, while physical provenance incurs a consistent approximately 23% space overhead. Again, augmentation is responsible for essentially all of the space overhead for logical provenance.

B. Tracing Time

We measured the time to trace output tuples through our workflow to input data set ClaimData, after the workflow was

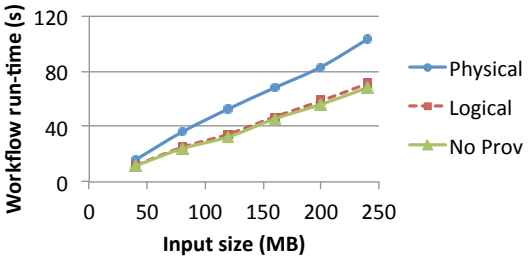

Fig. 7: Time overhead of provenance capture.

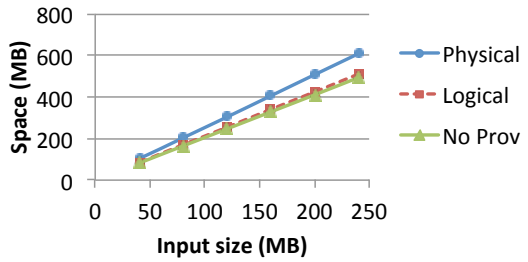

Fig. 8: Space overhead of provenance capture.

run collecting physical or logical provenance. We report two times for logical provenance (Figure 9):

1. **Logical:** Time to trace one output tuple using Panda's tracing algorithm
2. **Logical Uncombined:** Time to trace one output tuple using a tracing algorithm that does not combine provenance across transformations

For our workflow, Panda's tracing algorithm combines provenance twice, eliminating two intermediate data sets from the tracing query (recall Section VII-C). Thus, while the tracing query for both physical provenance and uncombined logical provenance involves six data sets, the tracing query for combined logical provenance involves only four data sets.

As shown in Figure 9, tracing using uncombined logical provenance is 6-16% slower than using physical provenance. While both approaches involve tracing queries with the same number of data sets, tracing physical provenance is faster, since the data tables used in the tracing query for logical provenance are larger than the physical provenance tables. However, combining logical provenance in Panda's tracing algorithm enables faster tracing; tracing using combined logical provenance is 32-36% faster than using physical provenance.

These experiments are preliminary, and our performance results only serve as one example of how logical provenance compares to physical provenance.

IX. Related Work

There has been a large body of work in lineage and provenance over the past two decades, surveyed in, e.g., [8], [12], [29]. As mentioned earlier, in contrast to our logical provenance, coarse-grained (schema and/or transformation level) provenance (e.g., [7], [19]) is usually insufficient to support element-level debugging and drill-down. Physical provenance can support debugging and drill-down, but requires each output

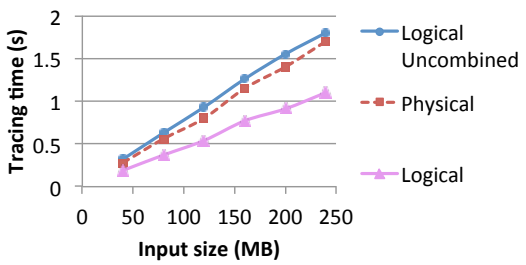

Fig. 9: Provenance tracing time.

element to be annotated with some type of identifier for the contributing input elements (e.g., [9], [26]).

Provenance models for relational and semistructured transformations are presented in, e.g., [5], [6], [10], [15], [16], [18], [25], [31]. Reference [10] contains a notion called *why-provenance*, which contains all *minimal witnesses* (combinations of input elements) that produce a given output element. Our notion of minimal provenance is a generalization of why-provenance; for the special case of monotonic transformations, minimal provenance is equal to the union of all minimal witnesses (Section II-C). In contrast to [15], our work defines notions of correctness and minimality for provenance that apply to general transformations, not just relational transformations. Reference [25] defines a notion related to provenance called *causality*, which captures not only the input elements that contribute to an output element, but also a measure of how responsible each input element is for producing the output element; the major results of [25] are restricted to conjunctive queries. Reference [6] considers the provenance of individual attribute values (*where-provenance*), while our work focuses on the provenance of data elements. Reference [16] captures provenance by augmenting relational transformations via query rewrite, in effect annotating output elements with provenance information. Our approach in some cases must resort to augmentation, but we attempt as much as possible to capture provenance information at the transformation level.

Provenance in the context of schema mappings is studied in, e.g., [13], [17], [31]. References [13] and [17] both consider schema mapping languages that cannot express general transformations (e.g., aggregation). Reference [31] considers a different type of provenance that consists of schema elements and transformations as opposed to input data elements.

Provenance in the data-oriented workflow setting is considered by [2], [3], [4], [11], [14], [20], [22], [23], [27], [32], among others. Reference [20] describes a demonstration scenario for the system presented in this paper, but does not provide any foundations, algorithms, or technical results. Reference [3] uses Pig Latin [28] to express the functionality of workflow modules, from which provenance information is generated, but unlike our general approach, provenance definitions are tightly coupled with the specific Pig Latin operations. Reference [11] presents techniques for reducing the space overhead of provenance storage, while our logical provenance uses transformation properties to avoid storing physical provenance altogether. Reference [32] requires the workflow creator to specify transformation *inverses* for each

transformation from which provenance can be computed; it has no support for automatically computing logical provenance for well-understood transformations, such as relational ones.

Reference [33] describes tracking provenance for arbitrary functions by instrumenting the program binary that implements the function. Since the approach in [33] involves capturing provenance based on a particular execution path, its definition of correct provenance may be missing relevant input elements. Also, by analyzing the properties of workflow provenance, we showed that an execution-based definition of provenance may include input elements that actually have no impact on the final output.

Reference [14] considers general transformations, providing a hierarchy of transformation types relevant to provenance; each transformation is placed in the hierarchy by its creator to make provenance tracing as efficient as possible. Provenance in [14] is defined separately for each transformation type, while the definitions for provenance given in this paper are unified across all transformations. Also, while [14] allows acyclic graphs of transformations, it does not investigate when properties such as correctness and minimality carry over from individual transformations to a composite workflow.

X. Conclusions

We presented a new general definition of provenance for transformations, introducing the notions of correctness, precision, and minimality. We determined when correctness, minimality, and weak correctness carry over from the individual transformations' provenance to the workflow provenance. We described a simple logical-provenance specification language consisting of attribute mappings and filters, and we provided a tracing algorithm in workflows where logical provenance for each transformation is specified using our language. We considered logical provenance in the relational setting, observing that for a class of SPJ transformations, logical specifications encode minimal provenance. We described our prototype system supporting the features and algorithms presented in the paper, and we reported a few experimental results.

One area of future work is to study the relationship between the precision and performance of tracing algorithms, perhaps identifying classes of workflows for which specialized tracing algorithms can greatly improve performance without losing much precision. Another area of future work is to explore other possible logical-provenance specification languages. Our language was designed to be expressive yet simple, but there may exist other specification languages that work especially well for workflows in particular domains.

Acknowledgment

We thank Parag Agrawal, Anish Das Sarma, and Alkis Polyzotis for many valuable discussions, and Junsang Cho, Charlie Fang, and Satoshi Torikai for coding contributions.

References

[1] http://www.heritagehealthprize.com.

[2] The Open Provenance Model — Core Specification (v1.1). Dec. 2009. http://eprints.ecs.soton.ac.uk/18332/.

[3] Y. Amsterdamer, S. B. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen. Putting lipstick on pig: enabling database-style workflow provenance. In *VLDB*, 2012.

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

[5] O. Benjelloun, A. Das Sarma, A. Halevy, and J. Widom. ULDBs: Databases with uncertainty and lineage. In *VLDB*, 2006.

[6] D. Bhagwat, L. Chiticariu, W.-C. Tan, and G. Vijayvargiya. An annotation management system for relational databases. In *VLDB*, 2004.

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

[8] R. Bose and J. Frew. Lineage retrieval for scientific data processing: a survey. *ACM Comput. Surv.*, 37(1), 2005.

[9] S. Bowers, T. M. McPhillips, and B. Ludäscher. Provenance in collection-oriented scientific workflows. *Concurrency and Computation: Practice and Experience*, 20:519–529, April 2008.

[10] P. Buneman, S. Khanna, and W.-C. Tan. Why and where: A characterization of data provenance. In *ICDT*, 2001.

[11] A. P. Chapman, H. V. Jagadish, and P. Ramanan. Efficient provenance storage. In *SIGMOD*, 2008.

[12] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4), 2009.

[13] L. Chiticariu and W.-C. Tan. Debugging schema mappings with routes. In *VLDB*, 2006.

[14] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. *The VLDB Journal*, 12(1), 2003.

[15] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM TODS*, 25(2), 2000.

[16] B. Glavic and G. Alonso. Perm: Processing provenance and data on the same data model through query rewriting. In *ICDE*, 2009.

[17] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update exchange with mappings and provenance. In *VLDB*, 2007.

[18] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, 2007.

[19] T. Heinis and G. Alonso. Efficient lineage tracking for scientific workflows. In *SIGMOD*, 2008.

[20] R. Ikeda, J. Cho, C. Fang, S. Salihoglu, S. Torikai, and J. Widom. Provenance-based debugging and drill-down in data-oriented workflows. In *ICDE*, 2012.

[21] R. Ikeda, A. Das Sarma, and J. Widom. Logical provenance in data-oriented workflows (long version). Technical report, Stanford University. http://ilpubs.stanford.edu:8090/1041/.

[22] R. Ikeda, H. Park, and J. Widom. Provenance for generalized map and reduce workflows. In *CIDR*, 2011.

[23] R. Ikeda, S. Salihoglu, and J. Widom. Provenance-based refresh in data-oriented workflows. In *CIKM*, 2011.

[24] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18:1039–1065, August 2006.

[25] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. The complexity of causality and responsibility for query answers and non-answers. In *VLDB*, 2011.

[26] P. Missier, K. Belhajjame, J. Zhao, M. Roos, and C. Goble. Data lineage model for Taverna workflows with lightweight annotation requirements. In *IPAW*, 2008.

[27] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

[28] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *SIGMOD*, 2008.

[29] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Rec.*, 34(3), 2005.

[30] J. D. Ullman. *Database and Knowledge-base Systems (Vol 2)*. Computer Science Press, 1989.

[31] Y. Velegrakis, R. J. Miller, and J. Mylopoulos. Representing and querying data transformations. In *ICDE*, 2005.

[32] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *ICDE*, 1997.

[33] M. Zhang, X. Zhang, X. Zhang, and S. Prabhakar. Tracing lineage beyond relational operators. In *VLDB*, 2007.